

Skkynet Embedded Toolkit for Renesas Synergy

User's Guide



This is the full documentation for the Skkynet Embedded Toolkit for Renesas Synergy, including an introduction to the ETK, and what you need to create and test a Renesas Synergy project, along with API and class reference information.

Copyright © 2016 Skkynet Cloud Systems, Inc. and its subsidiaries and licensors. All rights reserved.

Skkynet and the Skkynet logo, SkkyHub are trademarks of Skkynet Cloud Systems, Inc. DataHub and WebView are trademarks used under license. Protected by U.S. and foreign patents. For terms and conditions of use and full intellectual property notices, see: <http://skkynet.com/legal/>

Table of Contents

Overview	1
Skkynet ETK Architecture	3
DataHub and SkkyHub	5
Using Cogent DataHub	5
Using SkkyHub	5
Using Them Together	6
Data Points	7
User Threads	8
Modbus Master Support	9
Getting Started	10
Creating a New Project	11
1) Create a Project	11
2) Configure the SSP	11
3) Configure the Skkynet ETK	13
4) Generate the project content	14
5) Configure the build environment	14
6) Build your project	16
Testing the Sample Application	17
1) Install and Configure Cogent DataHub	17
2) Configure a DataHub Connection	17
3) Configure a SkkyHub Connection	18
4) Next Steps	21
Customizing Your Application	22
Application Mainline	22
Template Files	22
ThreadX Memory Usage	22
API	25
Data Quality Values	25
Modbus Addressing	25
Multi-threaded API	27
Typedef Documentation	28
Function Documentation	28
ThreadX Memory Usage	35
Define Documentation	36
config_app.c	38
Function Documentation	39
config_app.h	39
Typedef Documentation	40
Function Documentation	40
Define Documentation	42
config_modbus.c	43
Typedef Documentation	44
Variable Documentation	44
Function Documentation	44
config_points.c	45
Variable Documentation	46
Function Documentation	46
config_threads.c	47
Variable Documentation	47

Function Documentation	47
config_timers.c	48
Variable Documentation	48
Function Documentation	48
mainline.c	49
Variable Documentation	50
Function Documentation	51
Define Documentation	55
Classes	56
Arg struct Reference	56
Buf struct Reference	56
BufferSpec struct Reference	56
Bytecode struct Reference	56
CAppConfig struct Reference	57
CBufferedSocket struct Reference	62
CCharBuffer struct Reference	63
CCommand struct Reference	63
CCommandList struct Reference	63
CConnectionFactory struct Reference	64
CDataHubPoint struct Reference	64
Cell struct Reference	64
CellList struct Reference	65
Cons struct Reference	65
CSocket struct Reference	66
CSortedPtrArray struct Reference	66
CTCPClient struct Reference	66
CTCPConnection struct Reference	67
CTCPConnectionContainer struct Reference	69
CTimer struct Reference	70
CWebSocketDecoder struct Reference	70
Environment struct Reference	71
ETK_Api struct Reference	71
EtkThread struct Reference	71
EtkThreadData struct Reference	72
EtkThreadStruct struct Reference	72
File struct Reference	72
Function struct Reference	73
GCContext struct Reference	73
Heap struct Reference	73
Instance struct Reference	74
Klass struct Reference	74
Lambda struct Reference	74
LispInterpreter struct Reference	75
LispTimer struct Reference	77
MessageQueue struct Reference	78
ModbusConnection struct Reference	78
ModbusDataType struct Reference	78
ModbusDataValue struct Reference	79
ModbusIoMap struct Reference	79
ModbusMessage struct Reference	79
ModbusMessageType struct Reference	80
ModbusPointRef struct Reference	80

ModbusPointSpec struct Reference	81
ModbusTransform struct Reference	81
PointerStack struct Reference	82
PrintContext struct Reference	82
PT_ChangeRequest struct Reference	82
PT_stCPOINT struct Reference	83
PT_uVALUE union Reference	83
Scope struct Reference	83
StackPosition struct Reference	83
StringStream struct Reference	83
Symbol struct Reference	84
SymbolMap struct Reference	84
ThreadMessage struct Reference	84
TryState struct Reference	84
Type struct Reference	85
UT_stCMD struct Reference	85
ValueStack struct Reference	85
Vector struct Reference	85
WriteContext struct Reference	86

Overview

The Skkynet Embedded Toolkit (ETK) is a C library that allows the developer to quickly create applications that can send and receive data in real time to both to the [Cogent DataHub®](#) industrial middleware application and to the [Skkynet™](#) cloud service.

Data Data is identified as (name, value, quality, timestamp) tuples, allowing your application and all cooperating applications to interact with your data by name rather than by hardware address.

Cogent DataHub Data from the ETK can be transmitted on your local LAN to the [Cogent DataHub](#) industrial middleware, which can automatically convert it to OPC, DDE, ODBC, E-mail, TCP, Modbus or custom formats. The Cogent DataHub can also trigger scripts and actions based on data changes from your application, and update information on any industrial HMI. In effect, with the DataHub your application immediately becomes a first-class participant in any industrial control system.

Skkynet In addition, your data can be transmitted via the Internet to the [Skkynet](#) cloud service, where it can be accessed remotely by any authorized user. This allows you to monitor and control your embedded device without presenting an attack surface to the Internet. The Skkynet service provides everything you need to not only connect your device, but also to create web-accessible graphical HMIs for your service engineers, analysts and end users.

Developer-friendly The Skkynet ETK provides a developer-friendly method to establish and maintain a TCP socket connection and set of data points, distributing changes to these data points among connected client and server applications. Developers using the ETK can also use a thread-safe API within the ETK to create their own processing threads that can write data and subscribe to data point changes from the ETK engine.

More Help If questions come up that are not covered by this documentation, please feel free to contact Skkynet at [our website](#), by email: info@skkynet.com, or by phone: +1 905 702 7851.

Features include:

- Full-time connectivity to the server for minimum latency
- Transfer latencies only microseconds above network ping time
- Event-driven communication - only data changes are transmitted
- Bi-directional communication, allowing both monitoring and control
- Publish/subscribe data model
- Server-side data discovery - no server configuration necessary
- Efficient structured text data format for low bandwidth usage
- Multiple ingoing and outgoing data sockets on a single thread
- Integrated timers with round-robin sharing with socket data
- Automatic resynchronization when connection is lost and recovered
- Automatic connection retries
- Runs on architectures with no floating point support
- Small footprint
- Thread-safe API for developer threads to emit and consume data
- Optional built-in WebSocket support for traversing proxies

- Optional built-in scripting for powerful local processing
- Optional support for SSL
- Optional support for IPV6
- Optional support for Modbus master to multiple slaves

Supports the following targets:

- Linux (ARM, x86)
- uClinux
- ThreadX (for Renesas Synergy)
- Windows (Cygwin)
- Windows (Visual Studio)
- QNX
- Portable to most platforms offering a BSD socket API

Skkynet ETK Architecture

The Skkynet ETK provides a simple API to transmit and receive data between the application and one or more data servers. The data servers can be [Cogent DataHub](#) for LAN and in-plant industrial use, or [SkkyHub](#) for cloud-based remote monitoring and control.

The data produced and consumed by the user application consists of [data points](#). Each data point represents a single sensor, actuator, computed value or information item. The value, quality and timestamp of a point can change at any time. Any change to value or quality is considered a significant event and should be *written* to the Skkynet ETK. This will result in a network message being sent to the server, and propagated to any clients that are connected and listening for changes to that data point.

The Skkynet ETK provides an optional [Modbus/TCP](#) layer that can be used to communicate with Modbus slave devices, and to translate the data from those devices into data events in the data server. This Modbus capability offers an alternative to data collection using directly connected sensors and transducers.

The core of the API is a single-threaded event loop, called the mainline thread ([mainline.c](#)), that waits for events from the following sources:

- incoming data from TCP connections
- a tick or an expiry from a user-specified timer
- a message arriving from another application thread

In theory any number of TCP connections can be open at once, allowing the application to transfer data to and from multiple servers simultaneously. In practice the number of threads is limited by available resources. In the NetX BSD socket implementation up to 32 concurrently open sockets are allowed.

The mainline thread configures the Synergy device and network, then enters an infinite loop. Within that loop, it calls **select** for all connected sockets, and waits up to a configurable **poll time** for incoming data. If no data arrives, it then proceeds to check for incoming messages from ETK-enabled application threads. All queued incoming messages are processed without waiting. Finally, it increments the tick for a set of user-defined timers, and calls the application callbacks when those timers expire.

All of the processing for these events occurs in the mainline thread. The developer can write application code to respond to these events in the mainline thread, or can trigger activity in [other application threads](#). The Skkynet ETK provides two mechanisms for safely crossing thread boundaries:

- An application thread can *register* for data change notifications from the mainline thread. The thread only registers for points that it is interested in, so different threads can see any subset of the data that they need. When a value changes in the mainline thread, whether from a remote source or from another thread, an event is queued to the application thread. The application thread must periodically service its queue to process incoming messages. The Skkynet ETK provides convenient functions for waiting on the queue and for traversing the events on the queue. If a point change notification is already enqueued to a thread when another change notification arrives for the same data point, the old notification is discarded and the new notification is placed at the end of the queue.

- The application can extend the Skkynet ETK message queue to add its own messages. These messages will not be discarded when new messages arrive, so an application should be careful to regularly process queued messages to avoid memory exhaustion, which would lead to a program failure.

The Skkynet ETK provides two levels of API. The lowest level consists of functions that can only be called in the mainline thread. This is effectively any public function in the Skkynet ETK. The second level consists of a thread-safe wrapper on some of these functions that can be called freely from within application threads.

The sample [mainline.c](#) file provides a complete implementation of the low-level API configuration and event loop. You may be able to use this mainline without modification in your application.

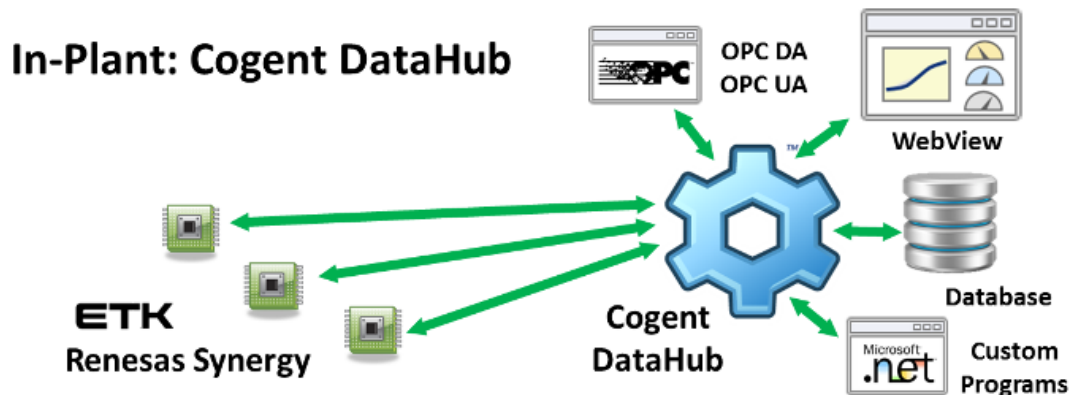
In order to use the [multi-threaded API](#), each thread must individually create an ETK object that acts as an identifier for the thread when calling the API functions. This encapsulated the thread's message queue, data point registrations and interface with the mainline thread.

DataHub and SkkyHub

The Skkynet ETK is designed to connect to the [Cogent DataHub®](#) software and the [SkkyHub™](#) cloud service.

Using Cogent DataHub

The [Cogent DataHub](#) is industrial middleware that accepts connections from the Skkynet ETK and integrate its data into any industrial control system. The DataHub provides bidirectional, real-time communication between OPC servers and clients, ODBC-compliant databases, Excel spreadsheets, Modbus devices, custom programs, and more. It can also trigger scripts and actions based on data changes from your application, and update information on any industrial HMI.

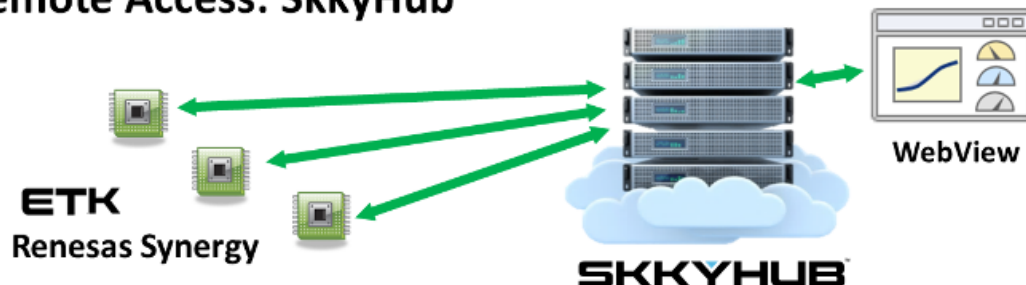


The Cogent DataHub runs on [any modern Windows platform](#). You can download a free demo version from the [Cogent DataHub](#) website. The [product documentation](#) provides complete instructions on how to use the DataHub, and a series of [how-to videos](#) help you get quickly up to speed.

Using SkkyHub

The [SkkyHub cloud service](#) allows you to securely monitor and control your embedded device from anywhere in the world in real time. The ETK's outbound-only connection architecture allows it to connect securely to SkkyHub, while presenting zero attack surface to the Internet. In this way, SkkyHub supports real-time M2M connectivity, as well as a web-based HMI that allows authorized access to application data.

Remote Access: SkkyHub

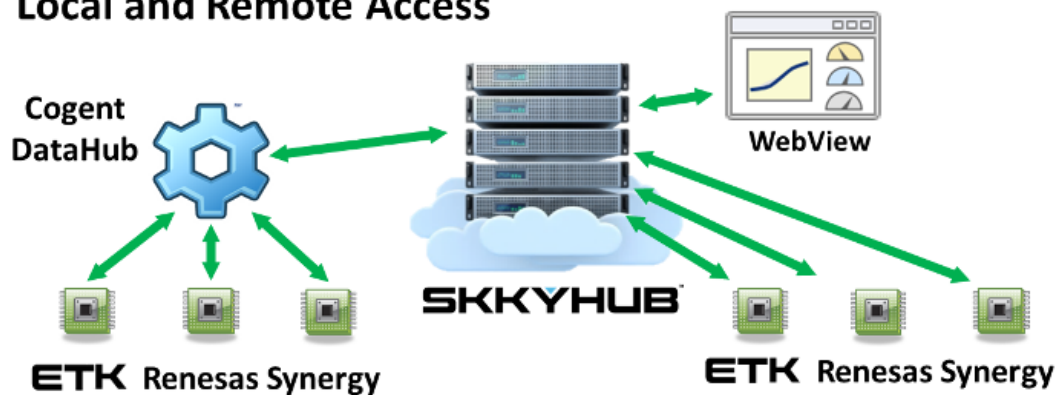


You can test a connection with SkkyHub in demo mode, as demonstrated in our [sample application test](#). To use SkkyHub with your application on an on-going basis, you will need a SkkyHub account. There are several different service types available to accommodate your needs, described in detail on the [SkkyNet website](#). The [SkkyHub documentation](#) provides the information you will need to [access the service](#), [administer your account](#), [use SkkyNet WebView](#), and more.

Using Them Together

The Cogent DataHub and SkkyHub are both fully compatible with each other, and with the SkkyNet ETK. Therefore you can use all of them in one fully integrated system.

Local and Remote Access



For example, the Cogent DataHub can be used to collect data from a number of devices running the ETK, and then send the consolidated data to SkkyHub. That data can be integrated with other data collected directly from the ETK, and any combination of it made available in WebView.

Data Points

The data produced and consumed by the user application consists of data points. Each point has an associated name, value, quality and timestamp. The name is the unique identifier for a point. In C code, if a pointer to the point structure is not available it can be looked up by name.

The value can be a 64-bit integer, a 64-bit float, or a UTF-8 encoded character string.

Quality is an indication of the reliability of the point value. These follow the OPC-DA data quality definitions for easy integration with industrial control systems.

Timestamp is stored in a type called #MSCLOCK, which is a pair of integer values representing the number of seconds since midnight January 1, 1970 UTC, and the number of nanoseconds since the beginning of the second, respectively. The application can set the timestamp if it has a valid time source, such as an NTP client or a GPS signal. If both of these values are zero, then it indicates to SkkyHub that it should stamp this data value with the server timestamp upon receipt. If possible, you should use a non-zero time here, as timestamping data values at the source is more accurate than using the server time, since it will not include a time offset due to network latency.

Communication between the server and the ETK names the point information about data points to a server, where the more complex types, such as arrays, are represented as strings.

See also: [Data Quality Values](#)

User Threads

The Skkynet ETK implements a single-threaded event loop that handles multiple TCP sockets and timers. In many cases, a developer may wish to implement his own code in a separate thread, and to interact with the main ETK thread via cross-thread messaging. The ETK main thread can communicate data change events to a user thread via a message queue, which the user thread then reads. In order to do this, the developer must implement an event loop within his thread that periodically processes queued events from the main thread.

Generally, the event loop runs forever until it is told to stop through a flag that indicates that it has been asked to terminate. At that point, the thread should clean up any resources that it holds and then simply return from its thread handler function.

It is up to the thread implementer to create the event processing loop. The ETK provides some functions that simplify this implementation, consisting of a simple test for thread termination and a pair of macros that loop through any pending messages for this thread. For example, a thread function might look like this:

```
ETK api = ETK_Init();

for (; !ETK_IsTerminating(api);)
{
    // Wait up to 10 msec for a message. You could replace this
    // with a Sleep(10) call if you need the timing for the
    // application-specific processing below.
    if (ETK_MessageWait(api, 10000) == 0)
    {
        ETK_FOREACH_MESSAGE(api, msg)
        {
            if (msg->type == ET_MSG_POINT_WRITE)
            {
                PT_ChangeRequest *cr = (PT_ChangeRequest*)msg;
                CDataHubPoint *point = cr->point;
                // Do some processing on point here, e.g., write to I/O device
            }
        }
        ETK_END_FOREACH(api, msg);
    }

    // Perform application-specific processing here, e.g., read I/O devices
}

ETK_Delete(api);
```

When you are working with a separate thread, you should limit your interaction to only the [multi-threaded API](#), as the other functions are not necessarily thread-safe. The intention is that you configure the communication with the data point server in the main thread, and only use your own thread to interact with the application [data points](#).

See also: [Multi-threaded API](#)

Modbus Master Support

The Skkynet ETK implements a Modbus/TCP master that can simultaneously connect to any number of Modbus slave devices. Each slave device is serviced by a separate thread to minimize the time spent during polling, even in large systems.

Since the Skkynet ETK operates on [data points](#) and Modbus operates on registers, the Modbus master implementation includes a facility for mapping between Modbus addresses and data points. This includes the ability to map multiple addresses to single data points (as in wide integer or floating point numbers), and to perform a linear transformation while both reading and writing the Modbus registers.

When a Modbus connection is initiated it enters an infinite loop that establishes a connection to the Modbus slave and then periodically polls the slave for each configured data value. At each polling cycle a set of Modbus commands is computed that will minimize the number of transactions to read all of the configured addresses. This will automatically use multi-register Modbus commands when necessary. Once the values are read the ETK will determine whether any of the Modbus registers has changed since the last polling cycle, and only generate a data change event if the Modbus register has changed.

If a deadband is configured for the data point then the data change event will only occur if the change exceeds the deadband. This can be used very effectively in systems with limited WAN bandwidth to maintain low latency (by polling frequently) and low WAN bandwidth usage (by only sending significant value changes). Deadbands are commonly used when monitoring a device that produces high-frequency jitter.

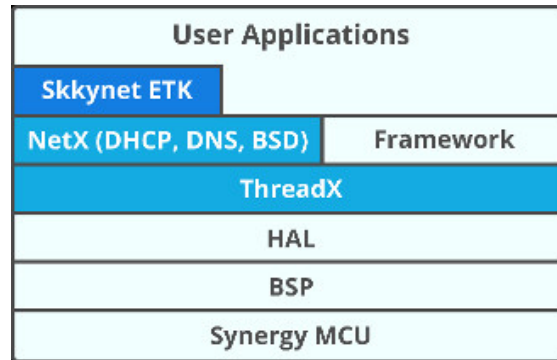
If both a deadband and a transform are configured for a point, then the deadband is computed in transformed units (engineering units) not in raw register values.

Any data point that is mapped to an output register or coil in the Modbus slave will be automatically written to the slave when a change event occurs on that point. You may mark an output register as read-only when creating the address mapping. In that case, changes to the data point value will not result in writes to the Modbus slave. Input registers are automatically considered read-only.

See also: [Modbus Addressing](#)

Getting Started

The Skkynet ETK is an application-level library that depends only on the ThreadX operating system, the NetX networking layer and an Ethernet network driver. Information about memory requirements is available in the section [ThreadX Memory Usage](#).



The Skkynet ETK is available for the Renesas Synergy embedded development platform:

http://am.renesas.com/products/embedded_systems_platform/synergy/index.jsp

To get started:

1. Install the e2 Studio development environment from the [Renesas Gallery web site](#).
2. Follow the instructions there to create a membership and to download the following:
 - a. e2 Studio (the development environment based on Eclipse, including an GNU ARM cross-compiler)
 - b. SSP (The Synergy Software Package, including support for a variety of hardware, ThreadX operating system, NetX network stack, etc.)
3. Once installed, download the Skkynet ETK installer from the Renesas gallery **Software Addons** section and run the installer on your computer. When asked for an installation path, please indicate the top-level directory where you have installed e2 Studio.

Now you are ready to [create a new project](#).

Creating a New Project

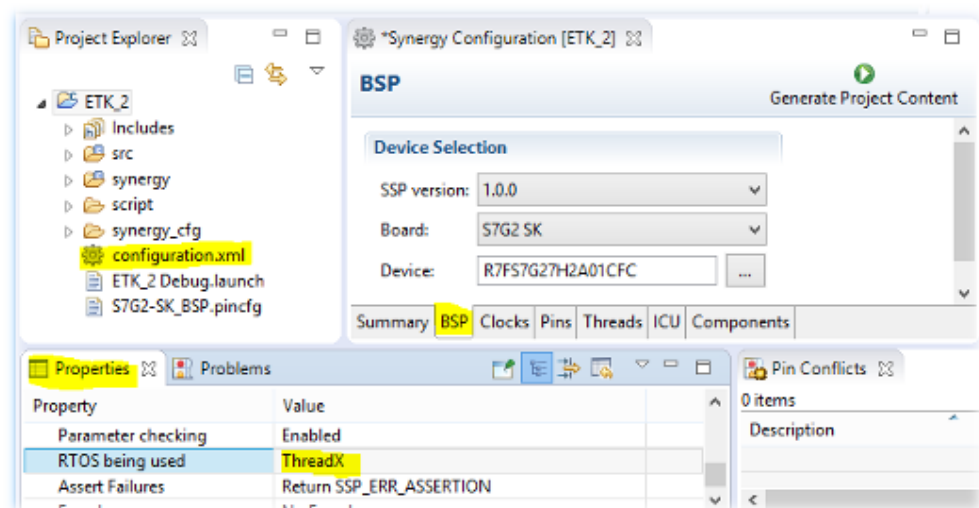
To create a Synergy project with the Skkynet ETK, follow these steps within e2 Studio:

1) Create a Project

1. Ensure that the Skkynet ETK is installed on your system.
2. Select **File # New # Synergy Project**.
 - a. Give your project a name.
 - b. Select your License file.
 - c. Press **Next**.
3. Select the target board:
 - a. Select the board (like **S7G2 SK**).
 - b. Press **Next**.
4. Select a project template:
 - a. Select a project that includes your BSP (like **S7G2-SK BSP**).
 - b. Press **Finish**.
5. Wait for the project to be created. You may get a message saying: "This kind of project is associated with the Synergy Configuration perspective. Do you want to open this perspective now?" Choose **YES** to go straight to the configuration, explained in the next step.

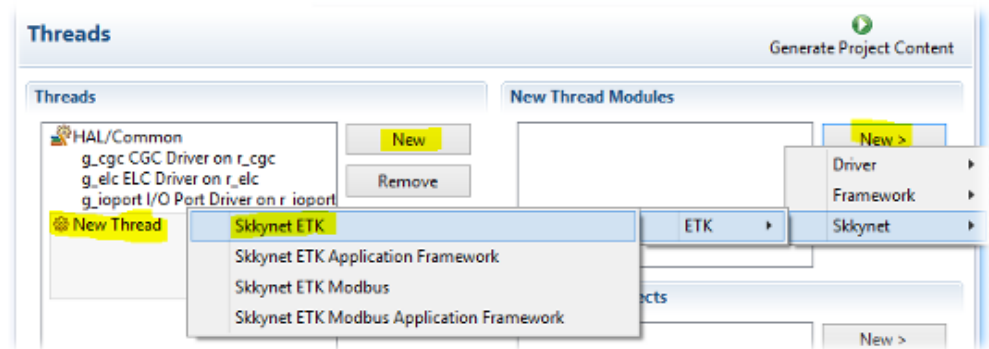
2) Configure the SSP

1. Double-click **configuration.xml** in the project source tree.
2. Configure the RTOS:
 - a. Select the **BSP** tab at the bottom of the **Synergy Configuration** pane.
 - b. Select the **Properties** tab at the top of the bottom pane of the e2 Studio window (beneath the Synergy Configuration pane).
 - c. Scroll to the bottom of the Properties tab and change the setting:
 - **RTOS being used to ThreadX**

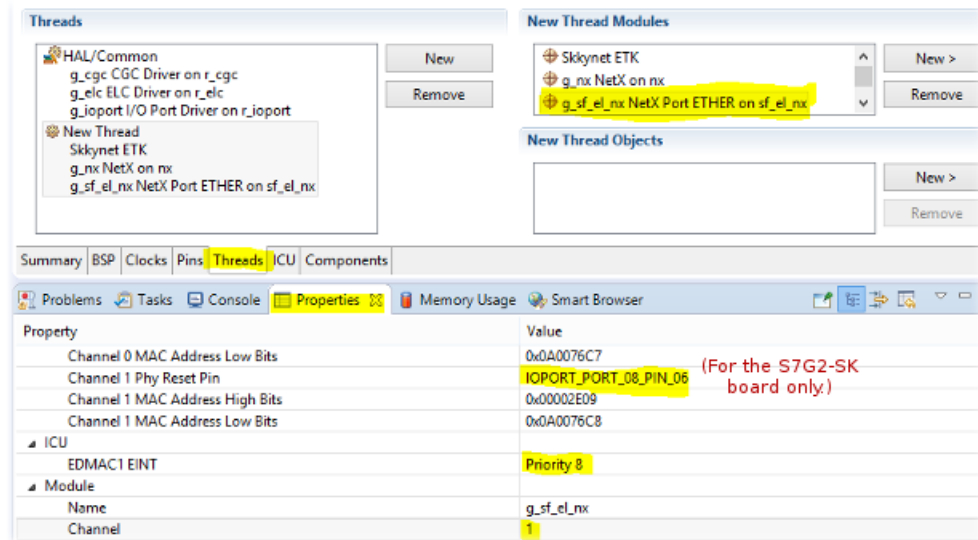


3. Create a thread:

- Select the **Threads** tab at the bottom of the Synergy Configuration pane.
- Press the **New** button in the center of the Threads pane.
- Press the **New >** button to the right of the New Thread Modules list:
 - Choose **Skkynet # ETK # Skkynet ETK**.

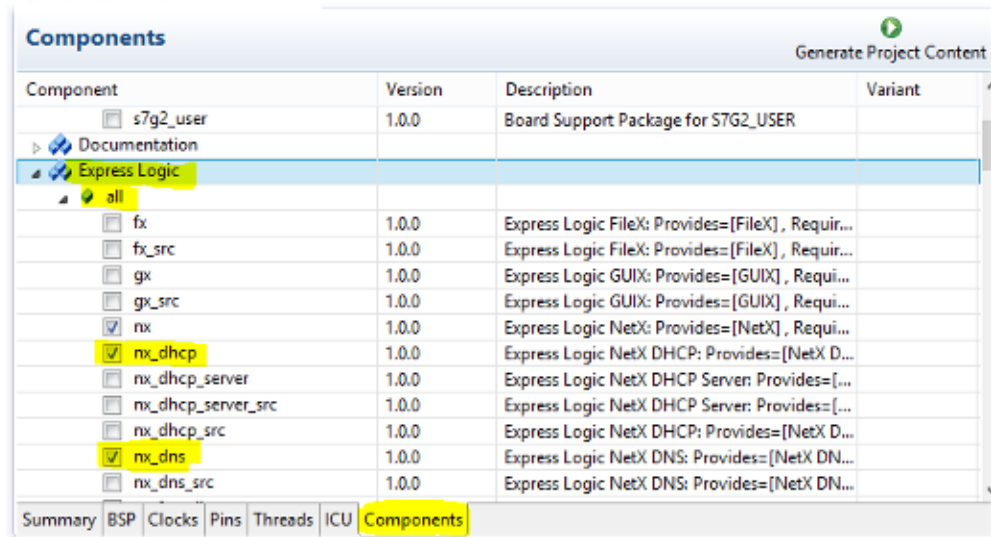


- Repeat the previous step for each of these, as needed:
 - SkkynetApp** (optional - contains application template source)
 - SkkynetModbus** (if you are using Modbus)
 - SkkynetAppModbus** (optional - contains application template source for Modbus)
- Select the **SkkynetETK** item in the **New Thread Modules** list.
 - Select the **Properties** tab at the top of the bottom pane of the e2 Studio window
 - If you do not wish to use DHCP in your application, set "Use DHCP" to "No"
 - If you do not wish to use DNS in your application, set "Use DNS" to "No"
 - If you wish to change the size of the Skkynet ETK heap, set it here. The size of the heap will depend on the number of data points that your application uses. You will need at least 32K. You can find more information about memory usage here: [ThreadX Memory Usage](#).
- In a similar way, press the **New >** button to the right of the New Thread Modules list, and choose:
 - Framework # Networking # NetX on nx** (required)
 - Framework # Networking # NetX Port ETHER on sf_el_nx**. (required)
- Select the **g_sf_el_nx NetX Port ETHER on sf_el_nx** item in the **New Thread Modules** list.
- Select the **Properties** tab at the top of the bottom pane of the e2 Studio window and change the settings:
 - EDMAC1 EINT** to **Priority 8** (any number should do)
 - Channel** to **1**
- If you are using the S7G2-SK board, select the **Properties** tab at the top of the bottom pane of the e2 Studio window and change the settings:
 - Channel 1 Phy Reset Pin** to **IOPORT_PORT_08_PIN_06**



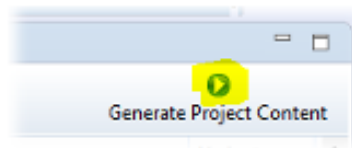
3) Configure the Skkynt ETK

1. Select the **Threads** tab at the bottom of the Synergy Configuration pane.
2. Select your newly created thread from the previous step.
3. Select **Skkynt ETK** in the New Thread Modules list:
 - Select the **Properties** tab at the top of the bottom pane of the e2 Studio window and change the setting:
 - **Modbus Support** to **0** or **1**, if you are using Modbus.
4. Select the **Components** tab at the bottom of the Synergy Configuration pane:
 - Ensure that the following components are selected:
 - **Express Logic # all # nx_dhcp**
 - **Express Logic # all # nx_dns**



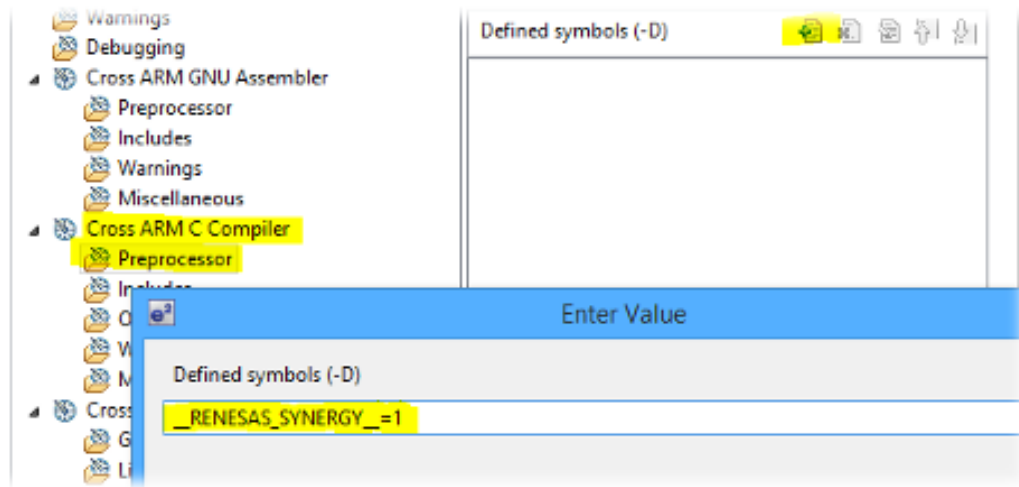
4) Generate the project content

- Press **Generate Project Content** in the top-right corner of the Synergy Configuration pane.

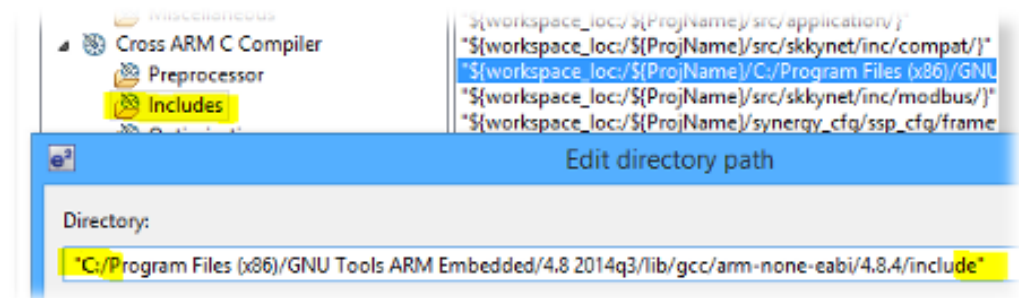


5) Configure the build environment

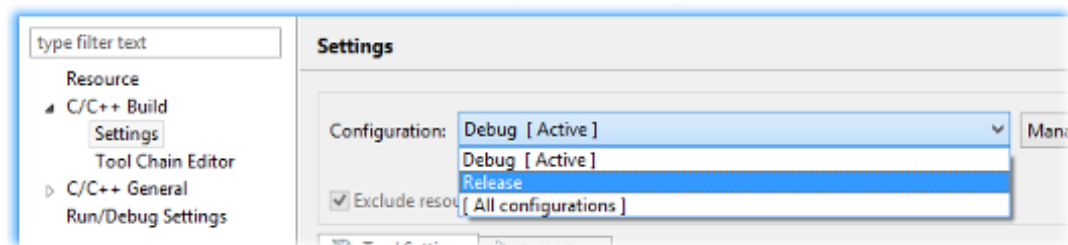
- Find your project in the e2 studio Project Explorer pane.
- Right-click the project title and select **Properties**.
- Expand **C/C++ Build** in the left-hand pane of the Properties dialog.
- Select **Settings** within **C/C++ Build**.
- Select **Cross ARM C Compiler # Preprocessor** in the right-hand pane (not Assembler or Linker).
- In the **Defined Symbols** section, add this symbol (starting and ending with two _ characters):
 - __RENESAS_SYNERGY__=1**



7. Select **Cross ARM C Compiler # Includes**.
8. Edit the include path beginning with `${workspace_loc}/${ProjName}/C:/Program Files (x86)/...` as follows:
 - Remove the string `${workspace_loc}/${ProjName}/` from the beginning of the path, leaving the opening double-quote.
 - Remove the character `}` from the end of the path, leaving the closing double-quote.



9. Repeat the modifications to the **Preprocessor** and **Includes** for the **Release** configuration, so that both **Debug** and **Release** configurations contain them.



10. Press OK to close the Properties dialog.

6) Build your project

- Right-click on your project title in the Project Explorer and select **Build Project**. If the Console output contains three lines similar to the following near the end, then your build was successful:

```
arm-none-eabi-size --format=berkeley "ETK_2.elf"
text data bss dec hex filename
136704 2336 123540 262580 401b4 ETK_2.elf
```

You should now have a project that implements a connection to a DataHub or SkkyHub server. You will need to configure the IP address or domain name, TCP port and data points for your application. If you have included Modbus/TCP support then you will need to configure the Modbus slave IP and the mapping between I/O addresses and point names. The sample files contain some examples of both single and multi-threaded operation, along with simple interaction with the LEDs on the target board. Please refer to the [Template Files](#) documentation for details.

Now you can [test the sample application](#).

Testing the Sample Application

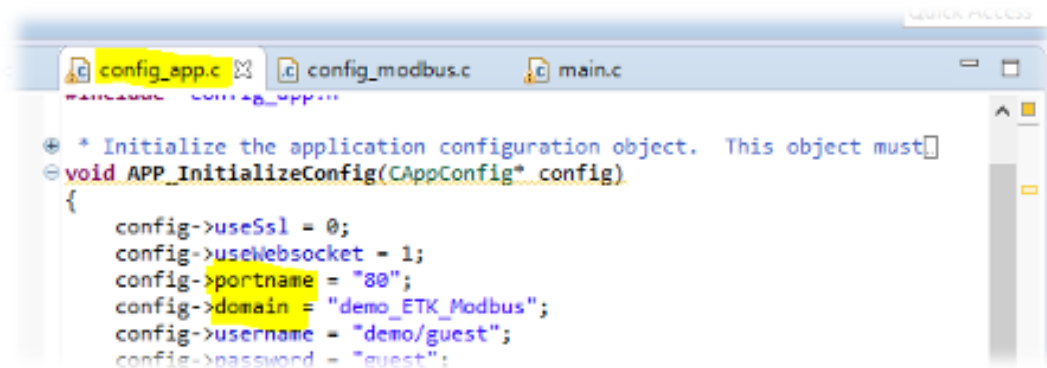
The sample application contains all of the required code to create a connection between your board and either a [Cogent DataHub®](#) or a [SkkyHub™](#) server. You will need to modify the configuration to match your application requirements. The simplest test is to install the Cogent DataHub on a PC on your local network, and to transmit data from your Synergy application to the DataHub. Once you are satisfied with your Synergy application you can sign up for an account on the SkkyHub cloud service and modify the target IP address and data domain in your application to send data to your cloud account.

1) Install and Configure Cogent DataHub

1. Download the Cogent DataHub from [the Cogent DataHub home page](#) and install it.
2. Configure the DataHub Web Server [as documented](#). Mainly, you need to ensure that the **Act as a web server** option is checked. You may also need to change the port number if your PC is running software that uses port 80 (e.g., Skype or IIS).

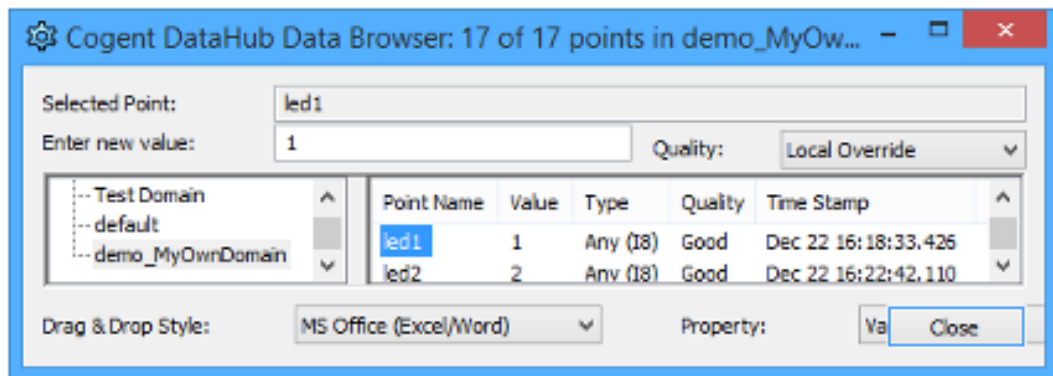
2) Configure a DataHub Connection

1. Open the file `src/application/config_app.c` in your e2 studio project.
2. Find the function `APP_InitializeConfig`, which sets the values of members of a `CAppConfig` structure.



3. Modify these values to match your network:
 - `hostname` - the IP address or host name of the computer running the DataHub
 - `portname` - the HTTP server port setting of the DataHub (normally 80)
 - `modbusHost` - the IP address or host name of the Modbus slave device
 - `modbusPort` - the port number of the Modbus slave device (normally 502)
4. In addition, you should modify the `domain` to create a data domain (essentially a namespace) for your data. Use the default prefix, `demo_` and change the string `ETK_Modbus` to a name of your choice, like this: `demo_Char1es_at_Acme` or similar. This name may be used later for testing on a public system, and you will get the best results if it is unique.
5. Open the file `src/application/config_app.h` in your e2 studio project.
 - If you are not using DHCP to assign an IP address, modify the definition for `STATIC_SERVER_IP_ADDRESS` to assign an IP address to this application.

- If you are not using DHCP, and you are using DNS, modify the definition for STATIC_DNS_SERVER_ADDRESS to the IP address of the DNS server. The address 8.8.8.8 is Google's public DNS.
 - If you are not using DHCP, and you are on a network with a router, modify the definition for STATIC_IP_GATEWAY_ADDRESS to the address of your router.
6. Rebuild and run your Synergy application
 7. Ensure that your Windows firewall settings allow an incoming connection on the DataHub HTTP server port ([portname](#) above).
 8. Start the Cogent DataHub, and click the **View Data** button to open the Data Browser window.



9. In the left pane, click the data domain for your application. Among the points listed, you should see at least led1 and led2.
 - The point led1 corresponds to LED 1 on your board. Depending on the board you are using this value will affect the LED differently. For example, on the SK-S7G2 a value of 0 will turn the light on, and non-zero will turn it off. On the DK-S7G2 a value of 0 will turn the LED off and values of 1 through 3 will select the LED color. You can change the value of the point by clicking on the name led1, and typing in a 1 in the **Enter a new value** field above, then pressing Enter.
 - The point led2 corresponds to LED 2 on your test board. Again, its behaviour depends on the board you are using.
 - Note: If you are using Modbus, you should see the default Modbus values appear in the Data Browser window as well.

If you are able to view and interact with your data in the Data Browser window, then you have successfully connected to the Cogent DataHub. Now you can continue, and configure a connection to SkkyHub.

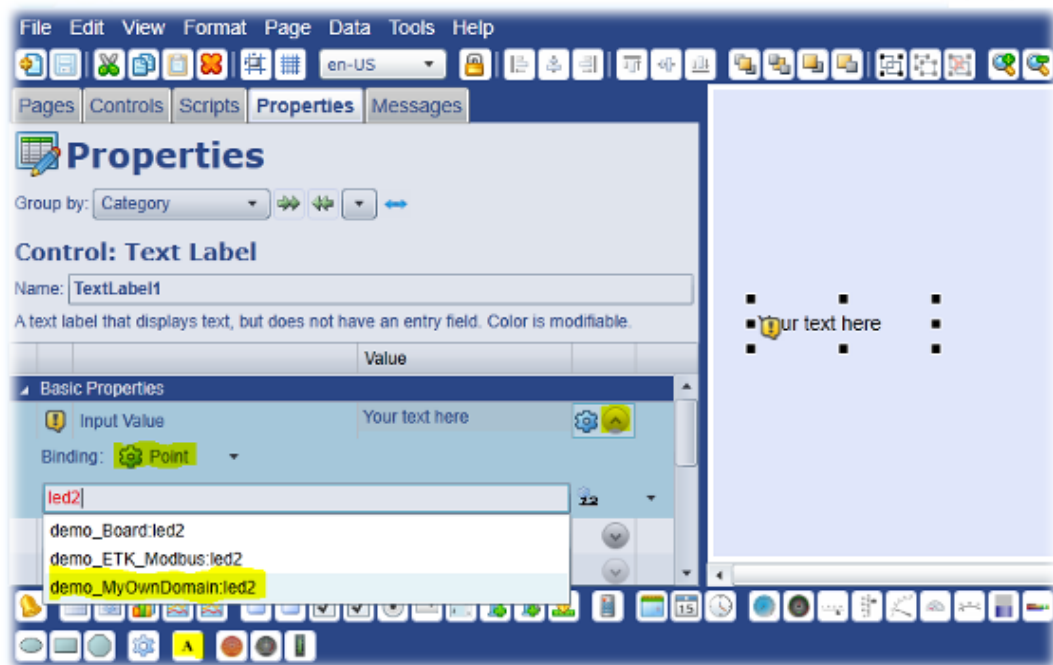
3) Configure a SkkyHub Connection

1. To connect to the SkkyHub service, open the file `src/application/config_app.c` in your e2 studio project.
2. Find the function `APP_InitializeConfig`, to set the values of the members of the `CAppConfig` structure.
3. Make the following changes:
 - `hostname` # `demo.skky.net.com`
 - `portname` # `80`

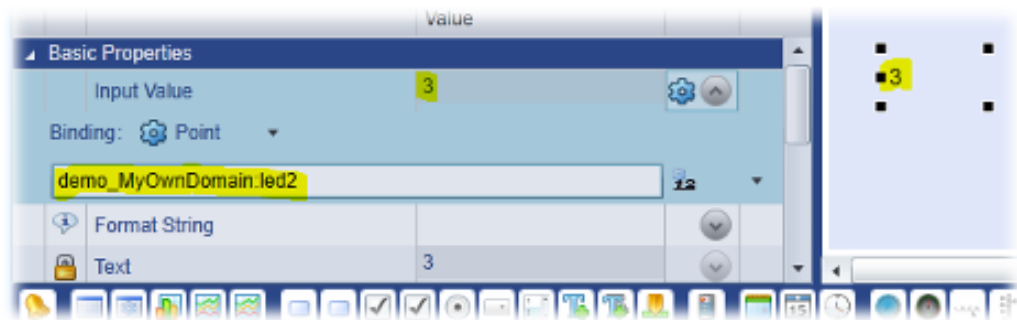
- **domain** # `demo_MyOwnDomain` (your domain name as specified previously)
- **username** # `demo/guest`
- **password** # `guest`

Note: If you already have a SkkyHub account, you can use your own user name, password, and data domain here. Otherwise, you need to use the guest account and `demo_your_domain` as described.

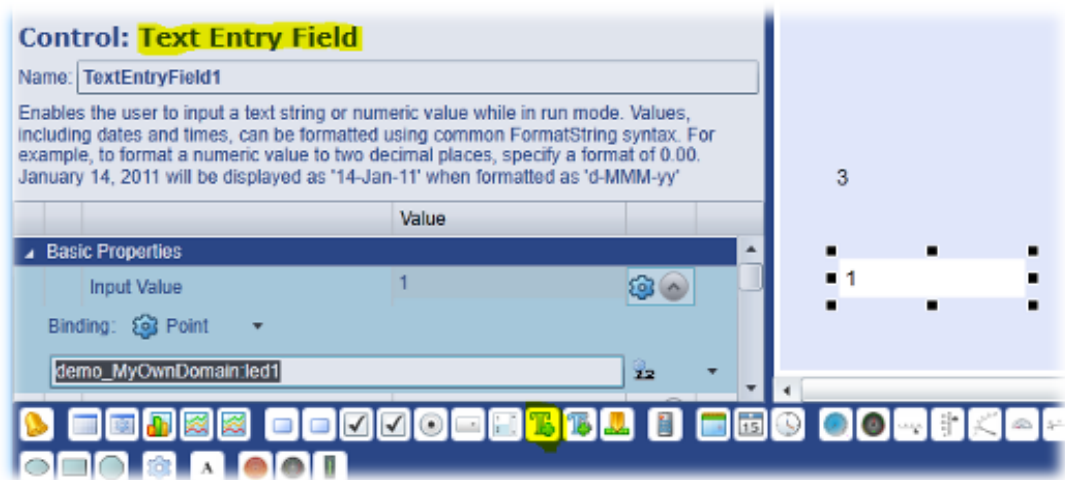
4. To check your data connection, open a browser and go to demo.skky.net.com.
5. Log in with the username Guest and password guest. This will open the SkkyNet WebView interface.
6. Select **File** # **New** to open a new page.
7. Add a new **Text Label control** by clicking the "A" button in the controls list at the bottom of the window.
8. In **Basic Properties** # **Input Value**, select the arrow button to open the Binding entry field.
9. Select **Point**, and in the entry field, type `led2`. This will search the domain for all points containing the string `led2`.
10. Choose the string `demo_yourdomainname:led2`.



The value of `led2` should appear in the control.



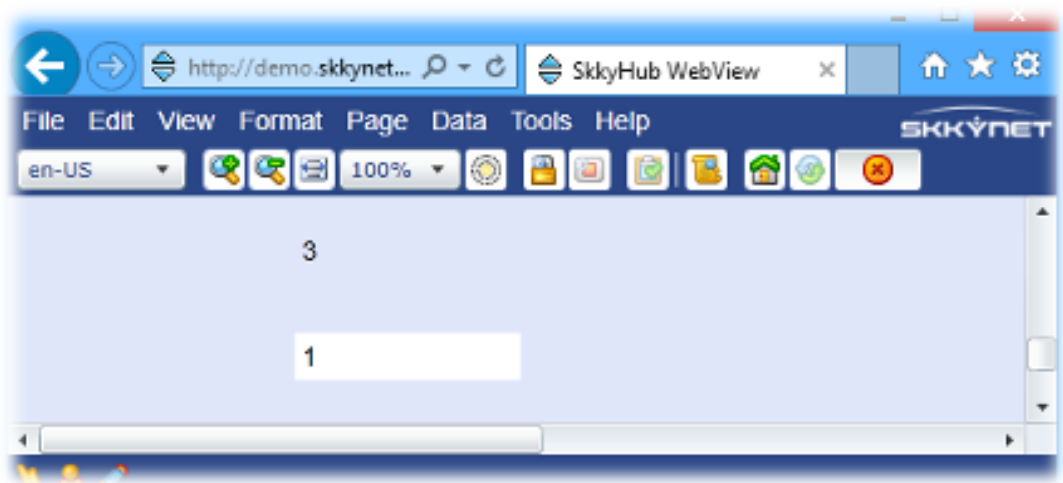
11. In the same way, add a new **Text Entry Field** control (a blue "T" with green plus sign) to the page, and link it to the data point `led1` in your data domain.



12. Click the green Run arrow at the top of the window.



This will put the page into Run mode, allowing you to change the value in the `led1` text entry field to demonstrate 2-way connectivity from SkkyHub to your device.



13. If you are able to see and interact with your data, then you have successfully connected to SkkyHub.

4) Next Steps

Now that you have tested your application with the DataHub and SkkyHub, you are ready to do any or all of the following, as explained in the relevant documentation:

- Configure more data points in [config_points.c](#).
- Configure Modbus connections and I/O mappings in [config_modbus.c](#).
- Configure user threads in [config_threads.c](#).
- Configure timers in [config_timers.c](#).
- [Customize your application](#).
- [Open a SkkyHub account](#). The demo account you have used for this test does not allow you to save pages or build an application. To do that, you will need a SkkyNet account.

Customizing Your Application

Application Mainline

The Skkynet ETK provides an optional ThreadX mainline ([mainline.c](#)) that you can use to automatically perform the following functions:

- Obtain an IP address using DHCP (this can be configured with an `#include` directive)
- Configure a DNS server from DHCP (this can be configured with an `#include` directive)
- Create a WebSocket connection to the target data server, either Cogent DataHub running on your local network, or SkkyHub running on the cloud
- Optionally create a Modbus connection to a Modbus slave device
- Automatically connect to DataHub or SkkyHub, and monitor the connection. If the connection fails, retry the connection indefinitely.
- Automatically connect to the Modbus slave, and monitor the connection. If the connection fails, retry the connection indefinitely.

Template Files

If you installed the SkkynetApp component then you will have a directory in your project called **src/application** that contains a number of files. These files provide the application mainline ([mainline.c](#)) and a number of files where you can add your application-specific functionality.

The application mainline is the entry point to your code during application start-up. It is responsible for setting up the NetX Ethernet driver, acquiring an IP address via DHCP, configuring a DNS server and starting an event loop that will periodically service any sockets and timers that you have set up during the initialization process.

During the initialization, the mainline will make calls to functions to configure SkkyHub/ DataHub data points, initiate timers and their handlers and configure the Modbus master parameters and Modbus I/O mappings if you have chosen to use Modbus. Examples of these functions are included in the templates: [APP_InitializeConfig](#), [APP_ConfigurePoints](#), [APP_ConfigureTimers](#), [APP_ConfigureUserThreads](#) and [APP_ConfigureModbus](#). With the exception of [APP_InitializeConfig](#), these functions are optional, and may be omitted from your application.

ThreadX Memory Usage

Summary of memory usage for the ThreadX O/S and Renesas Synergy.

Macro	Description
APP_HeapSize	The APP_HeapSize is used to determine the amount of memory to reserve for user thread stacks, internal structures, inter-thread messages and CDataHubPoint structures. Your application will consume memory from this heap depending on the number of connections, threads, points and timers that are configured. During development you can look at the performance of the heap by examining

	the global variables ME_Total, ME_Hiwater, ME_Nallocs, ME_Nfrees, ME_Nreallocs and ME_Hiaddress. Each user thread requires a stack whose size is defined by ETK_THREAD_STACK_SIZE. Each data point requires approximately 500 bytes. This varies with the length of the point name. Each CTimer requires approximately 80 bytes. Each inter-thread message in flight requires 56 bytes, plus the length of a string if the message contains a point change notification for which the value is a string type. We need to configure the heap size with a constant because we need to reserve heap early in the initialization process, before we have set the application configuration. This means that we cannot configure the application heap size dynamically from a configuration file. With Renesas Synergy, you can override the default heap size in the properties list of the Skkynet ETK module.
BSD_THREAD_STACK_SIZE	The NetX BSD layer uses a separate thread to handle select and poll calls. The default stack for this thread is 1024 bytes.
DHCP_THREAD_STACK_SIZE	The DHCP set-up uses a separate thread to obtain DHCP information. The default stack for this thread is 1024 bytes.
DNS_THREAD_STACK_SIZE	The DNS set-up uses a separate thread to configure the DNS. The default stack for this thread is 1024 bytes.
ETK_THREAD_STACK_SIZE	Each user thread, including the thread for Modbus communication, requires its own stack space, which is allocated from the heap (see APP_HeapSize). This stack size is fixed at 2048 bytes. Consequently, you must size the heap according to the number of threads you expect to have in your application.
MAIN_STACK_SIZE	The Skkynet ETK uses its main thread to run an event loop that services all of the connections to the application, and to

	service any user threads that are using the ETK. The default stack size is 4096 bytes.
MAX_PACKETS	Sets the number of packets in the NetX packet pool
NETX_DRIVER_STACK_SIZE	The NetX driver uses a separate thread to manage IP traffic. The default stack for this thread is 1024 bytes.
PACKET_POOL_SIZE	NetX uses a fixed-size packet pool to maintain IP packets in-flight. This is allocated explicitly and provided when NetX is initialized. The default size is 50 packets, with each packet being 1624 bytes, for a total of 80 KB.
TICK_STACK_SIZE	The Skkynet ETK uses a thread to count clock ticks to use as a time base for its timers. The default size of the tick stack is 512 bytes.

See also: [ThreadX Memory Usage](#)

API

Data Quality Values

Detailed Description

This is a table of the possible data quality values, and their respective C symbols. All data qualities are considered to be bad except for PT_QUALITY_GOOD and PT_QUALITY_LOCAL_OVERRIDE.

C-Symbol	Value
PT_QUALITY_BAD	0
PT_QUALITY_UNCERTAIN	0x40
PT_QUALITY_GOOD	0xc0
PT_QUALITY_CONFIG_ERROR	0x4
PT_QUALITY_NOT_CONNECTED	0x8
PT_QUALITY_DEVICE_FAILURE	0xc
PT_QUALITY_SENSOR_FAILURE	0x10
PT_QUALITY_LAST_KNOWN	0x14
PT_QUALITY_COMM_FAILURE	0x18
PT_QUALITY_OUT_OF_SERVICE	0x1c
PT_QUALITY_WAITING_FOR_INITIAL_DATA	0x20
PT_QUALITY_LAST_USABLE	0x44
PT_QUALITY_SENSOR_CAL	0x50
PT_QUALITY_EGU_EXCEEDED	0x54
PT_QUALITY_SUB_NORMAL	0x58
PT_QUALITY_LOCAL_OVERRIDE	0xd8

See also [Data Points](#)

Modbus Addressing

Detailed Description

A complete Modbus I/O mapping consists of the following pieces of information:

- **ModbusConnection** - The Modbus slave connection object for this point
- **slaveId** - The slave ID for this point. This can be zero (0) to use the default slave ID for the modbus connection, or a number between 1 and 254 to target a specific modbus slave.
- **block** - The modbus I/O block, one of:
 - MB_DI - digital input
 - MB_DO - digital output (coil)
 - MB_AI - analog input (input register)
 - MB_AO - analog output (holding register)

- **type** - The data type
 - One of
 - i2, i4, i8 - 2, 4 or 8-byte integer
 - r4, r8 - float or double
 - b - digital (1-bit)
 - Integer and real types can be followed with a dot (.) and a set of flags:
 - d - swap Dwords in 8-byte types
 - w - swap Words in 4- and 8-byte types
 - b - swap Bytes in 2-, 4- and 8-byte types
 - i - swap bits in 2-, 4- and 8-byte types (not implemented)
 - - - treat integer types as signed instead of unsigned
 - r - read-only - do not allow output even on output types.
- **svrtype** - The data type to output to the DataHub or SkkyHub server. This is specified the same as 'type' above, but may not contain a .flags suffix.
- **address** - The offset within the memory address range for this type, as a string. This is a zero-based address. In some device documentation the addresses are one-based, so be sure to subtract 1 from the address in that case.
 - Integer types can be followed with a dot (.) and a bit field specifier:
 - n - nth bit in the integer, from zero
 - n-m - nth to mth bits inclusive in the integer, from zero
 - Bit types can be specified as:
 - n-m - create an integer from the bits in the range of address n to m.
- **pointname** - The name of the DataHub point, without a domain name
- **xform** - A transformation to apply to the value from the Modbus device. Can be one of:
 - XFORM_TYPE_DIRECT - No transformation.
 - XFORM_TYPE_LINEAR(*m*,*b*) - Linear transformation as in $Y = mX + b$. Note that *m* cannot be zero.
 - XFORM_TYPE_RANGE(*mbMin*,*mbMax*,*svrMin*,*svrMax*,*clampMin*,*clampMax*) - Range transformation. Parameters are:
 - *mbMin* - The minimum expected value from the Modbus device
 - *mbMax* - The maximum expected value from the Modbus device
 - *svrMin* - The minimum expected value from the DataHub/SkkyHub server
 - *svrMax* - The maximum expected value from the DataHub/SkkyHub server
 - *clampMin* - 0 for no clamp, 1 to limit values to $\geq mbMin$ when writing to the Modbus device
 - *clampMax* - 0 for no clamp, 1 to limit values to $\leq mbMax$ when writing to the Modbus device
- **deadband** - a deadband in transformed units to apply when reading from the Modbus device.
 - A value will not be sent to the server if $\text{abs}(\text{value} - \text{previous_value}) < \text{deadband}$.
 - A value of 0 indicates no deadband.

4- and 8-byte values are stored in two and four adjacent analog input or output values respectively, and converted during the read and write stages. Floating point numbers are stored in the byte representation of IEEE 4-byte floating point format, in either normal or inverted byte order.

See also [Modbus Master Support](#)

Multi-threaded API

Detailed Description

The Skkynet ETK provides a multi-threaded API that exposes a subset of the ETK's capabilities to a user thread. The functions in this API start with the prefix "ETK_". Functions that are not part of this API may not be thread-safe.

Classes

- struct [EtkThreadStruct](#)

Typedefs

- typedef struct EtkThreadStruct [EtkThreadStruct](#)

Functions

- void [ETK_Apilnit](#) ([CTCPConnectionContainer](#) * cc, [CTCPClient](#) * client)
- ETK [ETK_Init](#) ()
- void [ETK_Delete](#) (ETK handle)
- int [ETK_EmitPoint](#) (ETK handle, [CDataHubPoint](#) * point, PT_uVALUE * value, PT_TYPE valueType, int32_t quality)
- int [ETK_EmitRegister](#) (ETK handle, [CDataHubPoint](#) * point)
- [CDataHubPoint](#) * [ETK_LookupPoint](#) (ETK handle, char * pointname)
- [CDataHubPoint](#) * [ETK_CreatePoint](#) (ETK handle, char * pointname, int flags)
- int [ETK_SetPointNameInt](#) (ETK handle, char * pointname, INT64 value, int32_t quality)
- int [ETK_SetPointInt](#) (ETK handle, [CDataHubPoint](#) * point, INT64 value, int32_t quality)
- int [ETK_SetPointNameDouble](#) (ETK handle, char * pointname, double value, int32_t quality)
- int [ETK_SetPointDouble](#) (ETK handle, [CDataHubPoint](#) * point, double value, int32_t quality)
- int [ETK_SetPointNameString](#) (ETK handle, char * pointname, char * value, int32_t quality)
- int [ETK_SetPointString](#) (ETK handle, [CDataHubPoint](#) * point, char * value, int32_t quality)
- char * [ETK_GetPointNameString](#) (ETK handle, char * pointname)
- char * [ETK_GetPointString](#) (ETK handle, [CDataHubPoint](#) * point)
- INT64 [ETK_GetPointNameInt](#) (ETK handle, char * pointname)

- INT64 [ETK_GetPointInt](#) (ETK handle, [CDataHubPoint](#) * point)
- double [ETK_GetPointNameDouble](#) (ETK handle, char * pointname)
- double [ETK_GetPointDouble](#) (ETK handle, [CDataHubPoint](#) * point)
- void [ETK_Free](#) (ETK handle, void * mem)
- int [ETK_MessageWait](#) (ETK handle, int usec)
- int [ETK_IsTerminating](#) (ETK handle)
- int [ETK_RegisterPointName](#) (ETK handle, char * pointname)
- int [ETK_RegisterPoint](#) (ETK handle, [CDataHubPoint](#) * point)
- static void [cbEtkMessageHandler](#) (EtkThread * thread, ThreadMessage * msg, void * userdata)
- int [ETK_HandleMessages](#) (ETK handle, ETK_MessageHandler handler)

Typedef Documentation

typedef struct EtkThreadStruct EtkThreadStruct

Function Documentation

void ETK_Apilnit (CTCPConnectionContainer *cc, CTCPClient *client)

Initialize the threaded API. This needs to be called once from the main thread with the connection container and the client connection to the remote DataHub. It sets internal globals that are subsequently provided to client threads when they call [ETK_Init](#).

cc	
client	

ETK ETK_Init ()

Initialize the threaded API for this thread. This needs to be called when a thread starts, before any calls the the threaded API.

Returns: A pointer to the API handle for this thread. Use this in all subsequent calls to the ETK interface.

void ETK_Delete (ETK handle)

Delete all of the resources associated with a threaded API handle. After this call, the API handle is invalid and further calls using it will crash your application.

handle	
--------	--

int ETK_EmitPoint (ETK handle, CDataHubPoint *point, PT_uVALUE *value, PT_TYPE valueType, int32_t quality)

Asynchronously emit a data point change to the main thread. This will eventually result in a change to the point value, but possibly not before this thread continues.

Your thread will not be able to immediately read the value of the point and expect to see the value just set. Your thread will only know with certainty that the point value has been set once it receives a value change notification.

If your thread is not registered for change notifications then it only knows that the value change will eventually occur in the main thread.

If another value is set on this point before the original value has been processed by the main thread, then the new assignment will overwrite the pending one, and the pending assignment will never occur.

To collect value change notifications for this point, register for value changes in your thread using [ETK_RegisterPoint](#) and then wait for changes using an event loop using [ETK_FOREACH_MESSAGE](#) and [ETK_IsTerminating](#).

You should not need to call this function directly. Instead, use [ETK_SetPointInt](#), [ETK_SetPointDouble](#) and [ETK_SetPointString](#).

handle	- The API handle
point	- A pointer to the point being changed
value	- A pointer to the value union
valueType	- The type of value in the value union
quality	- The quality to apply to this change

Returns: 0 if this write has been queued, 1 if an existing queued write was replaced by this one, and -1 if the write was aborted.

int ETK_EmitRegister (ETK handle, CDataHubPoint *point)

Emit an asynchronous message to register this thread for change notifications for this point. The thread will not receive a notification for this point's current value. It will only receive notifications for any changes to the point after this call has been processed. It is possible that the point may change value between the time that your thread makes this call and the time that the main thread processes it. No notification will be sent to your thread for such a change.

You should not need to call this function directly. Instead, use [ETK_RegisterPoint](#).

handle	- The handle to the ETK API structure
point	- The point to register

Returns: 0 on failure, 1 on success

CDataHubPoint* ETK_LookupPoint (ETK handle, char *pointname)

Look up a point structure by name. This will return a pointer to the point structure. The pointer will never be destroyed, but the information in it may change at any time in a thread-unsafe manner. Specifically, you should not rely on the string value of the point to be valid in your thread. To use the string value of a point you must call [ETK_GetPointString](#), which will make a copy of the string value in a thread-safe manner. When you are done using the value you must free it with [ETK_Free](#).

handle	- The handle to the ETK API structure
pointname	- The name of the point to find.

Returns: - A pointer to the point if it exists, or NULL.

CDataHubPoint* ETK_CreatePoint (ETK handle, char *pointname, int flags)

Create a new data point. If the flags include #PT_FLAG_WRITABLE then also register this thread for change notifications for this point. See [ETK_RegisterPoint](#) for more information on change notifications.

handle	- The handle to the ETK API structure
pointname	- The name of the point to create
flags	- One of #PT_FLAG_READABLE, #PT_FLAG_WRITABLE or #PT_FLAG_READWRITE

Returns: A pointer to the new point on success, or NULL.

int ETK_SetPointNameInt (ETK handle, char *pointname, INT64 value, int32_t quality)

Assign an integer value to a data point. See [ETK_SetPointInt](#) for more information.

handle	- The handle to the ETK API structure
pointname	- The name of the point to set
value	- The new value
quality	- The new quality for this value

Returns: -1 if the value was not queued, 0 if the value was newly queued, or 1 if the value replaces a value that was already queued.

int ETK_SetPointInt (ETK handle, CDataHubPoint *point, INT64 value, int32_t quality)

Assign an integer value to a data point. This will free any memory associated with the existing value and then replace it with the provided integer value.

This function will return immediately, before the value is actually written to the point. See [ETK_EmitPoint](#) for more information about the asynchronous behaviour of this function.

handle	- The handle to the ETK API structure
point	- The point to set
value	- The new value
quality	- The new quality for this value

Returns: -1 if the value was not queued, 0 if the value was newly queued, or 1 if the value replaces a value that was already queued.

int ETK_SetPointNameDouble (ETK handle, char *pointname, double value, int32_t quality)

Assign an double-precision floating point value to a data point. See [ETK_SetPointDouble](#) for more information.

handle	- The handle to the ETK API structure
pointname	- The name of the point to set
value	- The new value
quality	- The new quality for this value

Returns: -1 if the value was not queued, 0 if the value was newly queued, or 1 if the value replaces a value that was already queued.

int ETK_SetPointDouble (ETK handle, CDataHubPoint *point, double value, int32_t quality)

Assign an double-precision floating point value to a data point. This will free any memory associated with the existing value and then replace it with the provided double value.

This function will return immediately, before the value is actually written to the point. See [ETK_EmitPoint](#) for more information about the asynchronous behaviour of this function.

handle	- The handle to the ETK API structure
point	- The point to set
value	- The new value
quality	- The new quality for this value

Returns: -1 if the value was not queued, 0 if the value was newly queued, or 1 if the value replaces a value that was already queued.

int ETK_SetPointNameString (ETK handle, char *pointname, char *value, int32_t quality)

Assign a string value to a data point. See [ETK_SetPointString](#) for more information.

handle	- The handle to the ETK API structure
--------	---------------------------------------

pointname	- The name of the point to set
value	- The new value
quality	- The new quality for this value

Returns: -1 if the value was not queued, 0 if the value was newly queued, or 1 if the value replaces a value that was already queued.

int ETK_SetPointString (ETK handle, CDataHubPoint *point, char *value, int32_t quality)

Assign a string value to a data point. This will free any memory associated with the existing value and then replace it with the provided string value.

This function will return immediately, before the value is actually written to the point. See [ETK_EmitPoint](#) for more information about the asynchronous behaviour of this function.

handle	- The handle to the ETK API structure
point	- The point to set
value	- The new value
quality	- The new quality for this value

Returns: -1 if the value was not queued, 0 if the value was newly queued, or 1 if the value replaces a value that was already queued.

char* ETK_GetPointNameString (ETK handle, char *pointname)

Retrieve a string representation of a point value. See [ETK_GetPointString](#) for more information.

handle	
pointname	

Returns:

char* ETK_GetPointString (ETK handle, CDataHubPoint *point)

Retrieve a string representing of a point value. If the point value is an integer or a floating point number then create a string representation of the value and return that. Floating point numbers are formatted using sprintf directive "%.20g".

This function always allocates memory for the returned string, even if the point value is currently a string. It is up to the caller to free this memory using [ETK_Free](#).

handle	- The handle to the ETK API structure
point	- The point whose value to retrieve

Returns: A pointer to a newly-allocated string on success, or NULL on failure

INT64 ETK_GetPointNameInt (ETK handle, char *pointname)

Retrieve an integer representing of a point value. See [ETK_GetPointInt](#) for more information.

handle	- The handle to the ETK API structure
pointname	- The name of the point whose value to retrieve

Returns:

INT64 ETK_GetPointInt (ETK handle, CDataHubPoint *point)

Retrieve an integer representing of a point value. If the point value is a floating point number, it is truncated and case to integer. If the value is beyond the range of an integer then the value of this function is undefined.

If the point value is a string then this function will attempt to interpret the string as a decimal number. If the string starts with the characters **0x** then it will interpret the string as a hexadecimal number.

If the point value cannot be interpreted as a number then this function will return 0.

handle	- The handle to the ETK API structure
point	- The point whose value to retrieve

Returns: An integer

double ETK_GetPointNameDouble (ETK handle, char *pointname)

Retrieve a floating point representing of a point value. See [ETK_GetPointDouble](#) for more information.

handle	- The handle to the ETK API structure
pointname	- The name of the point whose value to retrieve

Returns: A double-precision floating point number

double ETK_GetPointDouble (ETK handle, CDataHubPoint *point)

Retrieve a floating point representing of a point value. If the point value is an integer, it is cast to a double. The result will be of the correct magnitude, but may lose precision during the translation.

If the point value is a string then this function will attempt to interpret the string as a floating point number. If the point value cannot be interpreted as a number then this function will return 0.

handle	- The handle to the ETK API structure
point	- The point whose value to retrieve

Returns: A double-precision floating point number

void ETK_Free (ETK handle, void *mem)

Free heap memory allocated by these functions: [ETK_GetPointString](#)
[ETK_GetPointNameString](#)

handle	- The handle to the ETK API structure
mem	- The memory address to be freed

int ETK_MessageWait (ETK handle, int usec)

Wait for a message to become available on the message queue associated with this thread. If no message is available within the specified number of microseconds, return. If *usec* is less than 0, wait forever. If *usec* is 0 then do not wait at all, simply returning an indication of whether a message is waiting.

handle	- The handle to the ETK API structure
usec	- The positive number of microseconds to wait, or negative to wait forever

Returns: 0 if there is a message waiting, or -1 if the wait timed out or failed.

int ETK_IsTerminating (ETK handle)

Determine whether the application has requested for this thread to terminate. Use this test to stop your thread's event loop.

handle	- The handle to the ETK API structure
--------	---------------------------------------

Returns: - non-zero if the thread has been asked to terminate, or zero otherwise

DllSym int ETK_RegisterPointName (ETK handle, char *pointname)

This function registers a point for notifications based on the point name. See [ETK_RegisterPoint](#) for more information.

This is an asynchronous call. It will return before the registration actually occurs. A success means that the request has been queued, not that the registration has completed.

handle	- The handle to the ETK API structure
--------	---------------------------------------

pointname	- The name of the point to register
-----------	-------------------------------------

Returns: 0 on failure, 1 on success

DllSym int ETK_RegisterPoint (ETK handle, CDataHubPoint *point)

This function registers a point for notifications, telling the ETK mainline that it should send a message to this thread whenever this point's value or quality changes. The ETK mainline will not send a message if the timestamp of the point changes, but the value and quality do not.

The notification will arrive at this thread via an asynchronous message that can be processed through the thread's [event loop](#).

This is an asynchronous call. It will return before the registration actually occurs. A success means that the request has been queued, not that the registration has completed.

handle	- The handle to the ETK API structure
point	- The point to register

Returns: 0 on failure, 1 on success

static void cbEtkMessageHandler (EtkThread *thread, ThreadMessage *msg, void *userdata)

int ETK_HandleMessages (ETK handle, ETK_MessageHandler handler)

This is an internal function that should not be called by user code.

handle	
handler	

Returns:

ThreadX Memory Usage

Brief Description

Summary of memory usage for the ThreadX O/S and Renesas Synergy.

Detailed Description

Defines

- #define [ETK_THREAD_STACK_SIZE](#) 2048
- #define [APP_HeapSize](#) 65536
- #define [TICK_STACK_SIZE](#) 512
- #define [MAIN_STACK_SIZE](#) 4096

- #define [MAX_PACKETS](#) 50
- #define [PACKET_POOL_SIZE](#) (([PACKET_PAYLOAD_SIZE](#) + sizeof([NX_PACKET](#))) * [MAX_PACKETS](#))
- #define [NETX_DRIVER_STACK_SIZE](#) 1024
- #define [DNS_THREAD_STACK_SIZE](#) 1024
- #define [DHCP_THREAD_STACK_SIZE](#) 1024
- #define [BSD_THREAD_STACK_SIZE](#) 1024

Define Documentation

#define [ETK_THREAD_STACK_SIZE](#)

Each user thread, including the thread for Modbus communication, requires its own stack space, which is allocated from the heap (see [APP_HeapSize](#)). This stack size is fixed at 2048 bytes. Consequently, you must size the heap according to the number of threads you expect to have in your application.

Definition at line 64 of file [et_threads.c](#)

The Documentation for this define was generated from the following file:

- [et_threads.c](#)

#define [APP_HeapSize](#)

The [APP_HeapSize](#) is used to determine the amount of memory to reserve for user thread stacks, internal structures, inter-thread messages and [CDataHubPoint](#) structures. Your application will consume memory from this heap depending on the number of connections, threads, points and timers that are configured. During development you can look at the performance of the heap by examining the global variables [ME_Total](#), [ME_Hiwater](#), [ME_Nallocs](#), [ME_Nfrees](#), [ME_Nreallocs](#) and [ME_Hiaddress](#).

Each user thread requires a stack whose size is defined by [ETK_THREAD_STACK_SIZE](#).

Each data point requires approximately 500 bytes. This varies with the length of the point name.

Each [CTimer](#) requires approximately 80 bytes.

Each inter-thread message in flight requires 56 bytes, plus the length of a string if the message contains a point change notification for which the value is a string type.

We need to configure the heap size with a constant because we need to reserve heap early in the initialization process, before we have set the application configuration. This means that we cannot configure the application heap size dynamically from a configuration file.

With Renesas Synergy, you can override the default heap size in the properties list of the Skkynet ETK module.

Definition at line 59 of file [config_app.h](#)

The Documentation for this define was generated from the following file:

- config_app.h

#define TICK_STACK_SIZE

The Skkynet ETK uses a thread to count clock ticks to use as a time base for its timers. The default size of the tick stack is 512 bytes.

Definition at line 104 of file mainline.c

The Documentation for this define was generated from the following file:

- mainline.c

#define MAIN_STACK_SIZE

The Skkynet ETK uses its main thread to run an event loop that services all of the connections to the application, and to service any user threads that are using the ETK. The default stack size is 4096 bytes.

Definition at line 114 of file mainline.c

The Documentation for this define was generated from the following file:

- mainline.c

#define MAX_PACKETS

Sets the number of packets in the NetX packet pool

Definition at line 121 of file mainline.c

The Documentation for this define was generated from the following file:

- mainline.c

#define PACKET_POOL_SIZE

NetX uses a fixed-size packet pool to maintain IP packets in-flight. This is allocated explicitly and provided when NetX is initialized. The default size is 50 packets, with each packet being 1624 bytes, for a total of 80 KB.

Definition at line 133 of file mainline.c

The Documentation for this define was generated from the following file:

- mainline.c

#define NETX_DRIVER_STACK_SIZE

The NetX driver uses a separate thread to manage IP traffic. The default stack for this thread is 1024 bytes.

Definition at line 143 of file mainline.c

The Documentation for this define was generated from the following file:

- mainline.c

#define DNS_THREAD_STACK_SIZE

The DNS set-up uses a separate thread to configure the DNS. The default stack for this thread is 1024 bytes.

Definition at line 151 of file mainline.c

The Documentation for this define was generated from the following file:

- mainline.c

#define DHCP_THREAD_STACK_SIZE

The DHCP set-up uses a separate thread to obtain DHCP information. The default stack for this thread is 1024 bytes.

Definition at line 159 of file mainline.c

The Documentation for this define was generated from the following file:

- mainline.c

#define BSD_THREAD_STACK_SIZE

The NetX BSD layer uses a separate thread to handle select and poll calls. The default stack for this thread is 1024 bytes.

Definition at line 167 of file mainline.c

The Documentation for this define was generated from the following file:

- mainline.c

config_app.c

Brief Description

Configure application-wide settings.

Detailed Description

Configure the application parameters here. If you have permanent storage, like a USB memory or access to the flash file system, load the configuration from there.

Functions

- void [APP_InitializeConfig](#) ([CAppConfig](#) * config)
- [CDataHubPoint](#) * [APP_CreateAndSetPoint](#) ([CTCPConnectionContainer](#) * cc, [CTCPClient](#) * client, char * name, int flags, INT64 value)
- void [APP_SetPointInt](#) ([CDataHubPoint](#) * point, INT64 value)
- void [APP_SetPointDouble](#) ([CDataHubPoint](#) * point, double value)

- void [APP_SetPointString](#) ([CDataHubPoint](#) * point, char * value)

Function Documentation

void [APP_InitializeConfig](#) ([CAppConfig](#) *config)

Initialize the application configuration object. This object must be allocated before entry to this function, and is typically created as a global static variable in the [mainline.c](#) mainline. You can modify this structure to suit your application requirements.

In the example code the contents of this object are hard-coded. In a production system you may prefer to store this information in persistent storage and then read it into the [CAppConfig](#) object here.

config	
--------	--

[CDataHubPoint](#)* [APP_CreateAndSetPoint](#) ([CTCPConnectionContainer](#) *cc, [CTCPClient](#) *client, char *name, int flags, INT64 value)

void [APP_SetPointInt](#) ([CDataHubPoint](#) *point, INT64 value)

void [APP_SetPointDouble](#) ([CDataHubPoint](#) *point, double value)

void [APP_SetPointString](#) ([CDataHubPoint](#) *point, char *value)

config_app.h

Brief Description

Define application-specific configuration.

Detailed Description

This file contains a definition of the [CAppConfig](#) structure that is passed to the application-specific configuration functions during application start-up. You can modify the [CAppConfig](#) structure to suit your application requirements.

Classes

- struct [CAppConfig](#)

Typedefs

- typedef struct [CAppConfig](#) [CAppConfig](#)

Functions

- [CDataHubPoint](#) * [APP_CreateAndSetPoint](#) ([CTCPConnectionContainer](#) * cc, [CTCPClient](#) * client, char * name, int flags, INT64 value)
- void [APP_SetPointInt](#) ([CDataHubPoint](#) * point, INT64 value)
- void [APP_SetPointDouble](#) ([CDataHubPoint](#) * point, double value)

- void [APP_SetPointString](#) ([CDataHubPoint](#) * point, char * value)
- void [APP_ConfigureTimers](#) ([CAppConfig](#) * config)
- void [APP_ConfigurePoints](#) ([CAppConfig](#) * config)
- void [APP_ConfigureModbus](#) ([CAppConfig](#) * config)
- void [APP_ConfigureUserThreads](#) ([CAppConfig](#) * config)
- void [APP_InitializeConfig](#) ([CAppConfig](#) * config)

Defines

- #define [APP_HeapSize_APP_HeapSize](#)
- #define [INIT_DNS](#) 1
- #define [INIT_DHCP](#) 1
- #define [INIT_GATEWAY](#) 1
- #define [STATIC_SERVER_IP_ADDRESS](#) IP_ADDRESS(192,168,0,2)
- #define [STATIC_SERVER_NETMASK](#) 0xFFFFFFFFUL
- #define [STATIC_DNS_SERVER_ADDRESS](#) IP_ADDRESS(8,8,8,8)
- #define [STATIC_IP_GATEWAY_ADDRESS](#) IP_ADDRESS(192,168,0,1)

Typedef Documentation

typedef struct [CAppConfig](#) [CAppConfig](#)

Defines an application-specific structure containing information that is required during start-up. In the example code, this information is supplied in config_app.c. In other instances this information might be stored as persistent configuration in flash memory. An application developer should add or remove members in this structure to suit the application requirements.

Function Documentation

[CDataHubPoint*](#) [APP_CreateAndSetPoint](#) ([CTCPConnectionContainer](#) *cc, [CTCPClient](#) *client, char *name, int flags, INT64 value)

void [APP_SetPointInt](#) ([CDataHubPoint](#) *point, INT64 value)

void [APP_SetPointDouble](#) ([CDataHubPoint](#) *point, double value)

void [APP_SetPointString](#) ([CDataHubPoint](#) *point, char *value)

void [APP_ConfigureTimers](#) ([CAppConfig](#) *config)

Configure the timers that will run in this application. Here you can create [CTimer](#) instances and supply them with tick and expiry callback functions that will run in the main ETK thread ([mainThreadEntry](#)). In the example code we create two timers, one to produce sample data at 10 Hz, and the other to produce a watchdog value at 1 Hz.

This is a good place to set up timers that will periodically read I/O registers and devices.

config	- A pointer to the application configuration object
--------	---

void APP_ConfigurePoints (CAppConfig *config)

Configure the data points that this application will use. Here you can create data points and supply them with change callback functions that will run in the main ETK thread ([mainThreadEntry](#)). In the example code we also create two points for two on-board LEDs, and a set of data points that can be randomly modified to provide a source of changing data during testing.

config	- A pointer to the application configuration object
--------	---

void APP_ConfigureModbus (CAppConfig *config)

Configures the Modbus connection.

This function opens a Modbus TCP connection and then configures it based on a mapping table between Modbus I/O addresses and data point names. It then starts the connection process, which will happen asynchronously.

In the sample code we use a static table to configure the Modbus I/O. Each entry in this table provides a mapping between a Modbus address and a DataHub / SkkyHub data point. The data point is identified by name, while the Modbus value is identified by its zero-based address. The data type of the Modbus value may be different from the data type transmitted to the DataHub. In addition, you can provide a linear transformation for each point so your application generates and consumes data in engineering units rather than raw transducer units. See [Modbus Addressing](#) for more information on Modbus addressing.

config	- The ETK application configuration object. In this implementation the Modbus slave address and port number are provided here.
--------	--

void APP_ConfigureUserThreads (CAppConfig *config)

Start any threads that will use the Skkynet ETK here. This is not strictly necessary, and you can eliminate this file from your application altogether. In that case, you will need to provide an alternate point of entry into your code, either through the [the mainline](#) or through one of the other config_* files.

config	- A pointer to the application configuration object
--------	---

void APP_InitializeConfig (CAppConfig *config)

Initialize the application configuration object. This object must be allocated before entry to this function, and is typically created as a global static variable in the [mainline.c](#) mainline. You can modify this structure to suit your application requirements.

In the example code the contents of this object are hard-coded. In a production system you may prefer to store this information in persistent storage and then read it into the [CAppConfig](#) object here.

config	
--------	--

Define Documentation

#define APP_HeapSize_

The number of bytes in the Skkynet ETK memory heap. See [APP_HeapSize](#) for more information.

Definition at line 66 of file [config_app.h](#)

The Documentation for this define was generated from the following file:

- [config_app.h](#)

#define INIT_DNS

Allow the mainline to assign the DNS server

Definition at line 116 of file [config_app.h](#)

The Documentation for this define was generated from the following file:

- [config_app.h](#)

#define INIT_DHCP

Allow the mainline to assign the IP address using DHCP

Definition at line 122 of file [config_app.h](#)

The Documentation for this define was generated from the following file:

- [config_app.h](#)

#define INIT_GATEWAY

Allow the mainline to assign the IP gateway

Definition at line 125 of file [config_app.h](#)

The Documentation for this define was generated from the following file:

- [config_app.h](#)

#define STATIC_SERVER_IP_ADDRESS

The static IP address if DHCP is not enabled

Definition at line 127 of file config_app.h

The Documentation for this define was generated from the following file:

- config_app.h

#define STATIC_SERVER_NETMASK

The netmask if DHCP is not enabled

Definition at line 128 of file config_app.h

The Documentation for this define was generated from the following file:

- config_app.h

#define STATIC_DNS_SERVER_ADDRESS

The DNS server address if DHCP is not enabled

Definition at line 129 of file config_app.h

The Documentation for this define was generated from the following file:

- config_app.h

#define STATIC_IP_GATEWAY_ADDRESS

The gateway address if DHCP is not enabled

Definition at line 130 of file config_app.h

The Documentation for this define was generated from the following file:

- config_app.h

config_modbus.c**Brief Description**

Configure connections and I/O mappings for Modbus slave connections.

Detailed Description**Classes**

- struct [ModbusPointSpec](#)

Typedefs

- typedef struct ModbusPointSpec [ModbusPointSpec](#)

Variables

- static ModbusPointSpec [PointSpecs](#)

Functions

- static int [mb_timer_handler](#) ([CTimer](#) * timer, void * userdata)
- static void [pointChangeHandler](#) ([CTCPConnectionContainer](#) * cc, [CTCPConnection](#) * writer, [CDataHubPoint](#) * point, void * data)
- void [APP_ConfigureModbus](#) ([CAppConfig](#) * config)

Typedef Documentation

typedef struct ModbusPointSpec ModbusPointSpec

Variable Documentation

ModbusPointSpec PointSpecs[]

Function Documentation

static int mb_timer_handler (CTimer *timer, void *userdata)

The ETK Modbus implementation uses a timer to poll the slave. This is the timer handler required to perform that poll.

timer	- A pointer to the CTimer object for this timer.
userdata	- User data provided when the timer was created. In this case it is a pointer to the ModbusConnection object.

Returns:

static void pointChangeHandler (CTCPConnectionContainer *cc, CTCPConnection *writer, CDataHubPoint *point, void *data)

A point change handler to act as an example. In this case, we do not need to do any extra processing when a point changes, as the ETK engine will handle sending any changes to writable Modbus registers when a change event occurs. The Modbus engine will deal with transformations between engineering and raw units.

cc	- A pointer to the CTCPConnectionContainer that holds this connection
writer	- A pointer to the CTCPConnection object that represents the connection that generated this change event

point	- The point that has changed.
data	- User data that was supplied when this change handler was configured.

void APP_ConfigureModbus (CAppConfig *config)

Configures the Modbus connection.

This function opens a Modbus TCP connection and then configures it based on a mapping table between Modbus I/O addresses and data point names. It then starts the connection process, which will happen asynchronously.

In the sample code we use a static table to configure the Modbus I/O. Each entry in this table provides a mapping between a Modbus address and a DataHub / SkkyHub data point. The data point is identified by name, while the Modbus value is identified by its zero-based address. The data type of the Modbus value may be different from the data type transmitted to the DataHub. In addition, you can provide a linear transformation for each point so your application generates and consumes data in engineering units rather than raw transducer units. See [Modbus Addressing](#) for more information on Modbus addressing.

config	- The ETK application configuration object. In this implementation the Modbus slave address and port number are provided here.
--------	--

config_points.c

Brief Description

Configure the data points that this application will process.

Detailed Description

Create data points here. We can declare a data point as either PT_FLAG_READABLE or PT_FLAG_READWRITE. If the point is PT_FLAG_READABLE then it is marked as read-only in the DataHub, meaning that it can only be changed by this server, and not by any client that is consuming it. If the point is PT_FLAG_READWRITE then the DataHub will show it as writable and a client application can change it. If a client changes a writable point then the change will propagate back to this application and we will have an opportunity to process the data change, for example by sending a message to the serial port. Marking a point as writable does not automatically register it for changes. That needs to be done on a per-connection basis.

Variables

- int [enableTestData](#)

Functions

- static void [hostHandshakeHandler](#) ([CTCPConnectionContainer](#) * cc, [CTCPConnection](#) * writer, [CDataHubPoint](#) * point, void * data)

- static void [ledHandler](#) ([CTCPConnectionContainer](#) * cc, [CTCPConnection](#) * writer, [CDataHubPoint](#) * point, void * data)
- void [APP_ConfigurePoints](#) ([CAppConfig](#) * config)

Variable Documentation

int enableTestData

Function Documentation

void hostHandshakeHandler ([CTCPConnectionContainer](#) *cc, [CTCPConnection](#) *writer, [CDataHubPoint](#) *point, void *data)

Handle onPointChange for the host handshake. When handshake_host changes to 1, invert handshake_client and set handshake_host back to 0.

cc	- The CTCPConnectionContainer that manages this data
writer	- A pointer to the client object that originated this data change
point	- The data point that changed
data	- User data supplied to CDataHubPoint_SetChangeHandler

void ledHandler ([CTCPConnectionContainer](#) *cc, [CTCPConnection](#) *writer, [CDataHubPoint](#) *point, void *data)

Handle onPointChange for the two LEDs. We are sending the LED number (0 or 1) in the user data field. When we receive a value, we write that to the LED I/O address. If the LEDs are 4-state (off, green, red, yellow) then write two bits from the incoming value, otherwise write one bit.

cc	- The CTCPConnectionContainer that manages this data
writer	- A pointer to the client object that originated this data change
point	- The data point that changed
vLedNum	- User data. In this case, the LED number (0 or 1) cast to void*

void APP_ConfigurePoints ([CAppConfig](#) *config)

Configure the data points that this application will use. Here you can create data points and supply them with change callback functions that will run in the main ETK thread ([mainThreadEntry](#)). In the example code we also create two points for two on-board LEDs,

and a set of data points that can be randomly modified to provide a source of changing data during testing.

config	- A pointer to the application configuration object
--------	---

config_threads.c

Brief Description

Configure user threads that will interact with the Skkynet ETK.

Detailed Description

Create your ETK-enabled threads here. You can perform any processing you like in these threads, so long as you periodically visit the ETK message queue to consume any data point changes sent to your thread. If your thread is a pure producer of data then it does not need to service the queue. The queue is designed to hold at most one message per data point, so if you do not process the queue in a reasonable time then stale values for a point will be discarded in favor of newer values. The message queue only guarantees that it will deliver the most recent value of a point to your thread.

Variables

- int [enableTestData](#)
- static TX_THREAD [testThread](#)

Functions

- static void [startAsyncTestThread](#) (CAppConfig * config)
- void [APP_ConfigureUserThreads](#) (CAppConfig * config)
- static void [test_thread_entry](#) (ULONG threadData)

Variable Documentation

int [enableTestData](#)

TX_THREAD [testThread](#)

Function Documentation

static void [startAsyncTestThread](#) (CAppConfig *config)

void [APP_ConfigureUserThreads](#) (CAppConfig *config)

Start any threads that will use the Skkynet ETK here. This is not strictly necessary, and you can eliminate this file from your application altogether. In that case, you will need to provide an alternate point of entry into your code, either through the [the mainline](#) or through one of the other config_*. files.

config	- A pointer to the application configuration object
--------	---

static void test_thread_entry (ULONG threadData)

config_timers.c

Brief Description

Configure single-threaded timers.

Detailed Description

Variables

- static char * [TestBanner](#)
- static int [TestIndex](#)

Functions

- static int [timer_handler](#) ([CTimer](#) * timer, void * userdata)
- static int [watchdog_handler](#) ([CTimer](#) * timer, void * userdata)
- void [APP_ConfigureTimers](#) ([CAppConfig](#) * config)

Variable Documentation

char* TestBanner[]

int TestIndex

Function Documentation

static int timer_handler (CTimer *timer, void *userdata)

static int watchdog_handler (CTimer *timer, void *userdata)

void APP_ConfigureTimers (CAppConfig *config)

Configure the timers that will run in this application. Here you can create [CTimer](#) instances and supply them with tick and expiry callback functions that will run in the main ETK thread ([mainThreadEntry](#)). In the example code we create two timers, one to produce sample data at 10 Hz, and the other to produce a watchdog value at 1 Hz.

This is a good place to set up timers that will periodically read I/O registers and devices.

config	- A pointer to the application configuration object
--------	---

mainline.c

Brief Description

Skkynet ETK mainline.

Detailed Description

This is the main entry point for a typical application using the Skkynet Embedded Toolkit. It provides the following:

Assignment of IP address using DHCP:

- requires the nx_dhcp component in your application
- #define INIT_DHCP in the file config_app.h to enable

Configuration of DNS server address:

- requires the nx_dnx component in your application
- #define INIT_DNS in the file config_app.h to enable
- If INIT_DHCP is also set, then use the DNS server returned from the DHCP server.

Initiate a mainline procedure that sets up the ETK and calls your functions to define timer handlers, data points and Modbus I/O mappings.

Variables

- static CAppConfig AppConfig
- TX_THREAD tick_thread
- TX_THREAD main_thread
- NX_PACKET_POOL pool_0
- NX_IP ip_0
- NX_DNS dns_0
- TX_THREAD dns_thread
- NX_DHCP dhcp_0
- TX_THREAD dhcp_thread
- ULONG error_counter
- static ULONG packet_pool_area
- static ULONG have_tick
- static ULONG entry_count

Functions

- VOID SYNERGY_ETHERNET_DRIVER (NX_IP_DRIVER *)

- static void `tickThreadEntry` (ULONG thread_input)
- void `mainThreadEntry` (ULONG thread_input)
- void * `init_malloc_memory` (void * first_unused_memory, int heapSize)
- INT `bsd_initialize` (NX_IP * default_ip, NX_PACKET_POOL * default_pool, CHAR * bsd_thread_stack_area, ULONG bsd_thread_stack_size, UINT bsd_thread_priority)
- void `dns_thread_entry` (ULONG i)
- void `dhcp_thread_entry` (ULONG i)
- VOID `hook_nx_ether_driver` (NX_IP_DRIVER * driver)
- void `tx_application_define_user` (void * first_unused_memory)
- static void `kick_timer` (ULONG ptr)
- static int `socket_close_timer_handler` (CTimer * timer, void * userdata)
- void `my_postConnectHook` (CTCPCClient * thisptr)
- void `APP_ConfigurePoints` (CAppConfig * config)
- void `APP_ConfigureTimers` (CAppConfig * config)
- void `APP_ConfigureModbus` (CAppConfig * config)
- void `APP_PointChangeHandler` (CAppConfig * config, CTCPCConnection * writer, CDataHubPoint * point)
- void `APP_ConfigureUserThreads` (CAppConfig * config)
- void `APP_InitializeConfig` (CAppConfig * config)
- void `mainline_onPointChange` (CTCPCConnectionContainer * thisptr, CTCPCConnection * writer, CDataHubPoint * point)

Defines

- #define `SERVER_IP_ADDRESS` IP_ADDRESS(0,0,0,0) /* Set the IP address to 0 for DHCP */
- #define `SERVER_NETMASK` 0x0UL /* Set the netmask to 0 for DHCP */
- #define `PACKET_PAYLOAD_SIZE` (1536 + 32)

Variable Documentation

CAppConfig AppConfig

A global variable containing the application configuration object. This is defined in `config_app.c` and `config_app.h`. You will need to modify `config_app.c` to set some connection information, and you may add extra members in `config_app.h` as necessary for your application.

TX_THREAD tick_thread

TX_THREAD main_thread

NX_PACKET_POOL pool_0

NX_IP ip_0

NX_DNS dns_0

TX_THREAD dns_thread

NX_DHCP dhcp_0

TX_THREAD dhcp_thread

ULONG error_counter

ULONG packet_pool_area[PACKET_POOL_SIZE/sizeof(ULONG)]

ULONG have_tick

This is a thread that just updates a global variable when a timer tick occurs. This is used to ensure that we only increment the ETK timers when a tick happens, rather than relying on select() to provide accurate timing.

ptr	- ignored
-----	-----------

ULONG entry_count

Function Documentation

VOID SYNERGY_ETHERNET_DRIVER (NX_IP_DRIVER *)

The Ethernet driver that your application uses will depend on the hardware. If there are multiple Ethernet interfaces then you must choose which one will be used for this application. This is set through a definition in config_app.h, and will typically be **nx_ether_driver_eth0** or **eth1**.

static void tickThreadEntry (ULONG thread_input)

void mainThreadEntry (ULONG thread_input)

This is the Skkynet ETK main thread function. It creates a container to hold data points, TCP connections, Modbus connections, timers and user thread information. It then triggers the connection process for TCP connections and enters an infinite loop where it continuously monitors sockets for data, handles asynchronous messages coming from user thread and manages timers.

thread_input	- ignored
--------------	-----------

void* init_malloc_memory (void *first_unused_memory, int heapSize)

INT bsd_initialize (NX_IP *default_ip, NX_PACKET_POOL *default_pool, CHAR *bsd_thread_stack_area, ULONG bsd_thread_stack_size, UINT bsd_thread_priority)

void dns_thread_entry (ULONG i)

This is the entry point for the DNS thread. The purpose of this thread is to create a DNS structure and then exit. If we are not using DHCP then we also declare the statically defined DNS server here.

i	- Thread start parameter. Ignored.
---	------------------------------------

void dhcp_thread_entry (ULONG i)

The purpose of this thread is to start the DHCP client and request an IP address. Loop indefinitely until the DHCP server responds with an address. Once the DHCP server responds, add the DNS entry from the DHCP server to the DNS control structre if we are using the DNS component.

i	- Thread start parameter. Ignored.
---	------------------------------------

VOID hook_nx_ether_driver (NX_IP_DRIVER *driver)

void tx_application_define_user (void *first_unused_memory)

This is the entry point from the Synergy start-up code. This is the first call that we should modify. The code that runs up to this point is auto-generated by the Synergy project generator.

Here we set up networking and start the DNS and DHCP configuration threads, then start the mainline thread that configures the Skkynet ETK and implements the event loop.

first_unused_memory	- passed by the system. We can create stack space and heap after this point.
---------------------	--

static void kick_timer (ULONG ptr)

static int socket_close_timer_handler (CTimer *timer, void *userdata)

This is a timer callback that will close a socket when the timer fires. Its purpose is to provide a transmission delay before dropping a socket to give the buffered data a chance to be transmitted.

timer	
-------	--

userdata	
----------	--

Returns: Always returns -1

void my_postConnectHook (CTCPClient *thisptr)

Override the onConnect event. This event is called when an outbound connection is successfully made. At thisptr point we may declare data points to the DataHub and register any data points that we want to receive from it.

thisptr	- Pointer to the TCP client structure that has just connected to its server.
---------	--

void APP_ConfigurePoints (CAppConfig *config)

Configure the data points that this application will use. Here you can create data points and supply them with change callback functions that will run in the main ETK thread ([mainThreadEntry](#)). In the example code we also create two points for two on-board LEDs, and a set of data points that can be randomly modified to provide a source of changing data during testing.

config	- A pointer to the application configuration object
--------	---

void APP_ConfigureTimers (CAppConfig *config)

Configure the timers that will run in this application. Here you can create [CTimer](#) instances and supply them with tick and expiry callback functions that will run in the main ETK thread ([mainThreadEntry](#)). In the example code we create two timers, one to produce sample data at 10 Hz, and the other to produce a watchdog value at 1 Hz.

This is a good place to set up timers that will periodically read I/O registers and devices.

config	- A pointer to the application configuration object
--------	---

void APP_ConfigureModbus (CAppConfig *config)

Configures the Modbus connection.

This function opens a Modbus TCP connection and then configures it based on a mapping table between Modbus I/O addresses and data point names. It then starts the connection process, which will happen asynchronously.

In the sample code we use a static table to configure the Modbus I/O. Each entry in this table provides a mapping between a Modbus address and a DataHub / SkkyHub data point. The data point is identified by name, while the Modbus value is identified by its zero-based address. The data type of the Modbus value may be different from the data type transmitted to the DataHub. In addition, you can provide a linear transformation for

each point so your application generates and consumes data in engineering units rather than raw transducer units. See [Modbus Addressing](#) for more information on Modbus addressing.

config	- The ETK application configuration object. In this implementation the Modbus slave address and port number are provided here.
--------	--

void APP_PointChangeHandler (CAppConfig *config, CTCPCConnection *writer, CDataHubPoint *point)

This function, if supplied, will be called in the mainline thread whenever a data point value or quality changes. You can use this to perform logging or other general point handling. Specific handlers can also be attached on a per-point basis.

This function will be called even if your own code changes the value, so be careful not to create infinite loops where you change a point in your code, then enter the point handler and change the point again.

config	- A pointer to the application configuration structure
writer	- A pointer to the connection that triggered this point change
point	- The point that changed

void APP_ConfigureUserThreads (CAppConfig *config)

Start any threads that will use the Skkynet ETK here. This is not strictly necessary, and you can eliminate this file from your application altogether. In that case, you will need to provide an alternate point of entry into your code, either through the [the mainline](#) or through one of the other config_*. files.

config	- A pointer to the application configuration object
--------	---

void APP_InitializeConfig (CAppConfig *config)

Initialize the application configuration object. This object must be allocated before entry to this function, and is typically created as a global static variable in the [mainline.c](#) mainline. You can modify this structure to suit your application requirements.

In the example code the contents of this object are hard-coded. In a production system you may prefer to store this information in persistent storage and then read it into the [CAppConfig](#) object here.

config	
--------	--

```
void mainline_onPointChange (CTCPConnectionContainer *thisptr, CTCPConnection *writer,
CDataHubPoint *point)
```

Define Documentation

#define SERVER_IP_ADDRESS

If you are using DHCP to assign an IP address to this device then the IP address and netmask should be set to zero here. If you are not using DHCP then you need to provide the information in config_app.h

Definition at line 53 of file mainline.c

The Documentation for this define was generated from the following file:

- mainline.c

Classes

Arg struct Reference

Public Attributes

- struct Cell * name
- struct Cell * dflt
- struct Cell * type
- int16_t flags
- int16_t _pad

Buf struct Reference

Public Attributes

- uint32_t allocated
- uint32_t len
- char * str

BufferSpec struct Reference

Classes

Public Attributes

- int address
- int addrlen
- int npoints
- uint8_t * bits
- uint16_t * words
- void * mem
- union BufferSpec::@0 @1

Bytecode struct Reference

Public Attributes

- struct Cell * code
- struct Cell * value

- struct Cell * stack
- int stackdepth

CAppConfig struct Reference

```
#include <config_app.h>
```

Public Attributes

- CTCPCConnectionContainer * cc
- CTCPCClient * client
- int useWebsocket
- int useSsl
- char * hostname
- char * portname
- char * domain
- char * username
- char * password
- int pwtype
- int pollUsecs
- int retrySecs
- int disconnectSecs
- MSCLOCK heartbeat
- MSCLOCK timeout
- MSCLOCK retry
- int heapBytes
- char * modbusHost
- char * modbusPort
- int modbusSlaveld
- int modbusPollMs
- ULONG macAddressLsw
- ULONG macAddressMsw

Detailed Description

Defines an application-specific structure containing information that is required during start-up. In the example code, this information is supplied in config_app.c. In other instances this information might be stored as persistent configuration in flash memory. An application developer should add or remove members in this structure to suit the application requirements.

Definition at line 77 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

Member Documentation

CTCPConnectionContainer* CAppConfig::cc

Needed by many ETK API calls

Definition at line 78 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

CTCPClient* CAppConfig::client

The client that connects to the DataHub or SkkyHub server

Definition at line 79 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

int CAppConfig::useWebsocket

0 or 1, indicating whether to connect using WebSocket

Definition at line 80 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

int CAppConfig::useSsl

0 or 1, indicating whether to use SSL (not implemented)

Definition at line 81 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

char* CAppConfig::hostname

The name of the DataHub/SkkyHub server

Definition at line 82 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

char* CAppConfig::portname

The port number (as a string) to connect to on the DataHub/SkkyHub server

Definition at line 83 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

char* CAppConfig::domain

The data domain into which data will be stored

Definition at line 84 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

char* CAppConfig::username

The user name for authentication on the server

Definition at line 85 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

char* CAppConfig::password

The password for authentication on the server

Definition at line 86 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

int CAppConfig::pwtype

The type of password encoding

Definition at line 87 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

int CAppConfig::pollUsecs

The number of microseconds per poll. ThreadX has a 10ms tick, so this should be a multiple of 10000

Definition at line 88 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

int CAppConfig::retrySecs

The number of seconds between socket connection re-tries when connecting to the server

Definition at line 89 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

int CAppConfig::disconnectSecs

The number of seconds to wait before disconnecting after a connection to the server is made. This has the effect of periodically connecting and updating data, then disconnecting again.

Definition at line 90 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

MSCLOCK CAppConfig::heartbeat

The keep-alive heartbeat rate

Definition at line 91 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

MSCLOCK CAppConfig::timeout

The keep-alive timeout. If no data or heartbeat is received from the server within this time, disconnect the socket.

Definition at line 92 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

MSCLOCK CAppConfig::retry

Internal. Use [retrySecs](#) instead.

Definition at line 93 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

int CAppConfig::heapBytes

Internal. Holds the configured heap size.

Definition at line 96 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

char* CAppConfig::modbusHost

The address of the Modbus slave device

Definition at line 99 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

char* CAppConfig::modbusPort

The port number (as a string) for the Modbus slave device

Definition at line 100 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

int CAppConfig::modbusSlaveId

The Modbus slave ID (1-254)

Definition at line 101 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

int CAppConfig::modbusPollMs

The number of milliseconds between Modbus polls

Definition at line 102 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

ULONG CAppConfig::macAddressLsw

The MAC address low 32 bits

Definition at line 105 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

ULONG CAppConfig::macAddressMsw

The MAC address high 16 bits. The high 16 bits of this value must be zero

Definition at line 106 of file config_app.h

The Documentation for this struct was generated from the following file:

- config_app.h

CBufferedSocket struct Reference

Public Attributes

- CSocket * m_Socket
- char * m_InBuf
- datalen_t m_InBufLen
- datalen_t m_InBufMaxLen
- datalen_t m_InBufLimit
- char * m_OutBuf
- datalen_t m_OutBufDataLen
- datalen_t m_OutBufDataStart
- datalen_t m_OutBufMaxLen
- datalen_t m_OutBufLimit
- datalen_t m_GrowRate
- datalen_t m_MessageLen
- UINT32 m_WebSocketFraming

- bool m_IsIncomingWebSocketClient
- bool m_IsOutgoingWebSocketClient
- CWebSocketDecoder * m_WebSocketDecoder
- UINT8 * m_WebSocketBuffer
- bool m_WebSocketCloseSent
- BOOL m_SomethingSent
- BOOL m_SomethingReceived
- BOOL m_DiscardNextMessage

CCharBuffer struct Reference

Public Attributes

- datalen_t m_GrowSize
- datalen_t m_MaxLength
- datalen_t m_Length
- char * m_Buffer

CCommand struct Reference

Public Attributes

- char * m_Name
- int m_Minargs
- int m_Maxargs
- UT_pfCMD m_Pfunc
- char * m_Description
- int32_t m_UserI1
- int32_t m_UserI2
- void * m_UserV1

CCommandList struct Reference

Public Attributes

- CSortedPtrArray * m_Commands

CConnectionFactory struct Reference

Public Attributes

- pfCFCreat pfCreateConnection

CDataHubPoint struct Reference

```
#include <Point.h>
```

Public Attributes

- PT_stCPOINT m_Point
- int m_ReadOnlyCount
- pfPointPointChange m_OnChange
- void * m_OnChangeData
- PT_ClientBits m_ClientRegistered
- PT_ClientBits m_ClientPending
- PT_ClientBits m_ClientSyncing

Detailed Description

A data point.

Definition at line 170 of file Point.h

The Documentation for this struct was generated from the following file:

- Point.h

Cell struct Reference

Classes

Public Attributes

- unsigned long type
- unsigned long inttype
- unsigned long symmap_resolved
- unsigned long mark
- unsigned long freed
- unsigned long destroying
- Symbol sym

- EX_LONG lval
- EX_REAL rval
- Cons cons
- Buf buf
- Klass * klass
- Instance * inst
- Function * func
- Bytecode * bytecode
- Vector * vector
- Vector * hash
- File * file
- Type type
- Environment env
- void * cvoid
- union Cell::@8 v

CellList struct Reference

Public Attributes

- Cell ** cells
- Cell * memory
- int maxcells
- int firstfree
- int deferred
- char locked
- char unsafe

Cons struct Reference

Public Attributes

- struct Cell * car
- struct Cell * cdr

CSocket struct Reference

Public Attributes

- SOCKET m_RawSocket
- CSocket_pfRecv m_pfReceive
- CSocket_pfSend m_pfSend
- SSL * m_SSL

CSortedPtrArray struct Reference

Public Attributes

- void ** m_Elements
- int m_NElements
- int m_MaxElements
- int m_GrowSize
- int m_IsUnsorted
- UT_pfCOMPARE m_CmpFunc
- ET_Mutex m_Mutex

CTCPClient struct Reference

Detailed Description

A [CTCPClient](#) is a connection from the ETK application to a server. That server can be the Skkynet Secure Cloud Service, Cogent DataHub running in MS-Windows, or Cascade DataHub running in Linux. Its purpose is to provide access to a set of data points that consist of a (name, value, timestamp, quality) tuple for each point. Whenever a data point changes in the server, it is optionally transmitted to the ETK application via the [CTCPClient](#) instance, and then retransmitted from the ETK application to other connected clients or servers, or transmitted to connected actuators, PLCs or other hardware.

[CTCPClient](#) connections can be either plain-text or SSL, depending on the user's choice and the availability of an SSL implementation on the embedded device.

[CTCPClient](#) connections can be via a direct socket connection, or via a WebSocket. If the connection is via a WebSocket then the connection can also specify a forward proxy server to use when making the connection. Proxies that require HTTP CONNECT are not supported.

[CTCPClient](#) derives from [CTCPConnection](#). All [CTCPConnection](#) functions will accept a [CTCPClient](#) argument.

Definition at line 15 of file TCPClient.c

The Documentation for this struct was generated from the following file:

- TCPClient.c

CTCPConnection struct Reference

Public Attributes

- SOCKET m_Socket
- int m_Port
- CBufferedSocket * m_BufSocket
- int m_ConnType
- bool m_IsBinary
- int32_t m_Flags
- int m_Errno
- int m_SSLErrno
- CTimer * m_HeartbeatTimer
- CTimer * m_TimeoutTimer
- char * m_DomainName
- MSCLOCK m_Heartbeat
- MSCLOCK m_Timeout
- bool m_ConnectInProgress
- struct CConnectionFactory * m_ConnFactory
- CCharBuffer * m_PendingToClient
- struct CTCPConnectionContainer * m_Container
- int m_Id
- CSortedPtrArray * m_PendingOutputPoints
- CTimer * m_ConnTimer
- char * m_Hostname
- char * m_Portname
- MSCLOCK m_RetrySecs
- MSCLOCK m_DisconnectSecs
- bool m_AutoRegisterDomain

- CSortedPtrArray * m_RegisteredPoints
- char * m_Username
- char * m_Password
- void * m_Userdata
- char * m_ProxyHostname
- char * m_ProxyPortname
- pfConnAttach Attach
- pfIntHandler0 cbSocketReadHandler
- pfIntHandler0 cbSocketWriteHandler
- pfIntHandler0 cbSocketExceptionHandler
- pfVoidHandler0 cbTimeoutHandler
- pfVoidHandler0 cbReconnectHandler
- pfVoidHandler0 CloseSocket
- pfVoidHandler0 SocketLostHandler
- pfVoidHandler0 onConnect
- pfVoidHandler0 preConnectHook
- pfVoidHandler0 postConnectHook
- pfVoidHandler0 onDisconnect
- pfVoidHandler0 preDisconnectHook
- pfVoidHandler0 postDisconnectHook
- pfVoidHandler0 postConnectionFailedHook
- pfVoidHandler0 onNewData
- pfVoidHandler0 onDelete
- pfPointWrite onPointWrite
- pfPointChange onPointChange
- pfCmdConnect onCmdConnect

Detailed Description

The [CTCPConnection](#) structure is the base structure for connections that can be represented as a file descriptor and managed using select, read, write, etc. This is generally TCP socket in any operating system with a BSD socket implementation, as well

as a serial connection on Un*x style systems like Linux. Both client and server connections use this structure as the basis for their implementations.

TCP socket connections must be made to the Skkynet Secure Cloud Service or to a Cogent DataHub running elsewhere. Serial connections are normally used to communicate with attached sensors or other equipment. Normally you should not create a [CTCPConnection](#) directly, but instead create a [CTCPClient](#).

Serial connections are implemented as [CTCPClient](#), with a different constructor. To create a serial client, call [CSerialClient_New](#). The result will be a [CTCPClient](#) pointer that talks to the serial port.

Definition at line 60 of file TCPConnection.h

The Documentation for this struct was generated from the following file:

- TCPConnection.h

CTCPConnectionContainer struct Reference

Public Attributes

- CSortedPtrArray * m_Connections
- CSortedPtrArray * m_Deletions
- CSortedPtrArray * m_Timers
- CSortedPtrArray * m_Points
- CCommandList * m_CommandList
- MSCLOCK m_TimerLastTime
- CTimer * m_CleanupTimer
- int m_IsTerminating
- int m_IsPreDeleted
- pfContainerPointChange onPointChange
- EtkThread * m_PseudoThread
- CTCPConnection * m_AllConnections[MAX_CLIENTS]
- ET_Mutex m_CreateMutex

Detailed Description

The [CTCPConnectionContainer](#) structure is the top-level construct for managing connections to TCP and serial clients and timers. Data points are managed from the connection container, and all connections in the container share the same data points. You must create one of these during your application initialization.

Definition at line 32 of file TCPConnectionContainer.h

The Documentation for this struct was generated from the following file:

- TCPConnectionContainer.h

CTimer struct Reference

```
#include <Timer.h>
```

Public Attributes

- int m_Active
- int m_AutoDelete
- int m_Deleted
- MSCLOCK m_Delay
- MSCLOCK m_Repeat
- MSCLOCK m_ExpiryTime
- void * m_Userdata
- UT_pfTIMER m_ExpireCallback
- UT_pfTIMER m_DeleteCallback
- UT_pfTIMERTICK m_TickCallback
- struct CTCPCConnectionContainer * m_TimerContainer

Detailed Description

A single-threaded timer based on the main event loop tick. You should not normally need to examine or modify the contents of this structure.

Definition at line 52 of file Timer.h

The Documentation for this struct was generated from the following file:

- Timer.h

CWebSocketDecoder struct Reference

Public Attributes

- int m_Protocol
- UINT64 m_PayloadLength
- UINT64 m_PayloadPosition
- int m_PayloadLengthFieldSize
- UINT8 m_MaskingKey[4]

- int m_IsFinal
- int m_Opcode
- int m_IsMasked
- UINT8 m_Frame[14]
- int m_CurrentFrameBytesReceived
- int m_CurrentFrameLength
- int m_CloseReceived
- int m_ControlFrameReceived

Environment struct Reference

Public Attributes

- struct Scope ** scopes
- unsigned short nscopes
- unsigned short maxscopes
- unsigned short temporary

ETK_Api struct Reference

Public Attributes

- CTCPCConnectionContainer * cc
- CTCPCClient * client
- CTCPCConnection * threadConnection
- EtkThread * thread

EtkThread struct Reference

Public Attributes

- MessageQueue * toThread
- MessageQueue * fromThread
- pthread_t id
- int err
- int isEnabled
- int isThreadActive

- struct EtkThreadData * data
- void * userdata

EtkThreadData struct Reference

Public Attributes

- pfThreadFunction startRoutine
- EtkThread * thread

EtkThreadStruct struct Reference

Public Attributes

- EtkThread * thread
- ThreadMessage * message
- void * userdata
- ETK_MessageHandler handler

File struct Reference

Public Attributes

- struct Cell * cname
- void * fptr
- void * lexbuf
- int32_t flags
- int32_t intdata
- int32_t linenum
- unsigned char ckey
- unsigned char cmask
- int16_t bufchar
- void * scanner
- void * scanner_state
- pfnOpen openfn
- pfnClose closefn
- pfnReopen reopenfn

- pfnFlush flushfn
- pfnAttach attachfn
- pfnPut putfn
- pfnGet getfn
- pfnUnget ungetfn
- pfnRead readfn
- pfnWrite writefn
- pfnWaiting waitingfn
- pfnSeek seekfn
- pfnTell tellfn
- pfnDestroyScanner destroyscannerfn

Function struct Reference

Public Attributes

- Lambda lambda
- char * cname
- struct Cell * lname

GCContext struct Reference

Public Attributes

- int denied
- void * heap
- int pargbase
- int pstacktop
- int eargbase
- int estacktop

Heap struct Reference

Public Attributes

- CellList ** lists
- int nlists

- int maxlists
- int lastfail
- int freecount
- int heapsize
- int lowlimit

Instance struct Reference

Public Attributes

- struct Cell * klass
- Buf rawdata
- int32_t flags
- struct Cell * vars

Klass struct Reference

Public Attributes

- struct Cell * name
- struct Cell * superclass
- struct Cell * cvars
- struct Cell * ivars
- struct Cell * localcvars
- struct Cell * localivars
- pfnClassConstructor c_constructor
- pfnClassDestructor c_destructor

Lambda struct Reference

Classes

Public Attributes

- uint8_t flags
- uint8_t minargs
- uint16_t maxargs
- Arg * args

- Arg rettype
- struct Cell * env
- EX_pflISPFN c
- struct Cell * l
- struct Cell * b
- union Lambda::@6 code

LispInterpreter struct Reference

Public Attributes

- uint32_t gc_denied
- uint32_t debug_info
- uint32_t use_signalheap
- uint32_t use_protectedheap
- uint32_t destroying
- uint32_t breaking
- uint32_t returning
- uint32_t throwing
- uint32_t runtime_type_checking
- short cellsize
- short blocksize
- Cell * nil
- Cell * truecell
- Cell * undef
- Cell * dot
- Cell * eof
- Cell * eol
- Cell * quote
- Cell * backquote
- Cell * comma
- Cell * commasplice

- Cell * let
- Cell * progn
- Cell * ifsym
- Cell * set
- Cell * setq
- Cell * setqq
- Cell * whilesym
- Cell * localsym
- Cell * eval
- Cell * defun
- Cell * defvar
- Cell * and
- Cell * or
- Cell * constructor
- Cell * destructor
- Cell * fstdin
- Cell * fstdout
- Cell * fstderr
- Cell * andtype
- Cell * andnoeval
- Cell * andoptional
- Cell * andrest
- Cell * andconst
- Cell * andexport
- Cell * andimport
- Cell * retval
- ValueStack env_stack
- ValueStack scope_cache
- Cell * current_env
- Cell * global_env

- Cell * global_symtab
- Cell * base_types
- Cell * base_type_names
- ValueStack variable_stack
- PointerStack protect_stack
- ValueStack vector_stack
- ValueStack vector_ind_stack
- ValueStack environment_cache
- ValueStack function_stack
- ValueStack error_stack
- ValueStack try_stack
- Cell * smallints
- int threadno
- struct Heap * currentheap
- struct Heap * mainheap
- struct Heap * signalheap
- struct Heap * protectedheap
- Cell * program_code
- long Allocations
- long SymbolEvals
- long SymmapEvals
- long SymmapLookups
- long FunctionEvals
- long OtherEvals
- Cell * currentFile
- int currentLine

LispTimer struct Reference

Public Attributes

- CTimer * timer

- Cell * expire_handler
- Cell * tick_handler

MessageQueue struct Reference

Public Attributes

- ET_Mutex mutex
- ThreadMessage * head
- ThreadMessage * tail
- int isEnabled
- int count
- pfThreadMessageDestructor destructor
- ET_Semaphore semHaveMessage

ModbusConnection struct Reference

Public Attributes

- modbus_t * modbus
- ModbusIoMap pointmaps[N_POINT_MAPS]
- void * userdata
- CModbusClient * client
- int isAsync
- EtkThread * thread
- CSortedPtrArray * messageTypes
- LispInterpreter * li
- int slaveid
- char * server
- char * port

ModbusDataType struct Reference

Public Attributes

- char baseType
- uint8_t length

- uint16_t flags

ModbusDataValue struct Reference

Classes

Public Attributes

- ModbusDataType type
- int8_t i1
- int16_t i2
- int32_t i4
- INT64 i8
- uint8_t ui1
- uint16_t ui2
- uint32_t ui4
- UINT64 ui8
- FLOAT r4
- DOUBLE r8
- uint8_t bytes[8]
- uint16_t words[4]
- uint32_t dwords[2]
- union ModbusDataValue::@2 value

ModbusIoMap struct Reference

Public Attributes

- int pointType
- CSortedPtrArray * pointrefs
- pfnModbusIoMapReader reader
- pfnModbusIoMapWriter writer

ModbusMessage struct Reference

Public Attributes

- ThreadMessage tm

- int result
- BufferSpec buf1
- BufferSpec buf2
- int bufferId
- int pointOffset
- Cell * script
- int replied
- int slaveid

ModbusMessageType struct Reference

Public Attributes

- int type
- pfSendMessageHandler sentHandler
- pfReplyMessageHandler replyHandler

ModbusPointRef struct Reference

Public Attributes

- uint8_t slaveid
- uint8_t blocktype
- uint16_t address
- int16_t isWriting
- int16_t isReading
- int16_t isInitialized
- int16_t isWaiting
- int16_t forceReadOnly
- ModbusDataValue lastValue
- ModbusDataValue waitingValue
- uint8_t bitstart
- uint8_t bitlength
- uint16_t arraylen

- ModbusDataType dataType
- ModbusDataType xformDataType
- ModbusTransform xform
- DOUBLE lastReadValue
- DOUBLE deadband
- CDataHubPoint * point

ModbusPointSpec struct Reference

Public Attributes

- int block
- char * modbusType
- char * remoteType
- int iaddress
- char * caddress
- char * pointName
- DOUBLE deadband
- pfPointPointChange changeHandler
- int xformType
- DOUBLE xformMultMbMin
- DOUBLE xformAddMbMax
- DOUBLE xformRemoteMin
- DOUBLE xformRemoteMax
- int xformClampMin
- int xformClampMax

ModbusTransform struct Reference

Classes

Public Attributes

- char type
- DOUBLE mult

- DOUBLE add
- struct ModbusTransform::@3::@4 linear
- DOUBLE mbmin
- DOUBLE mbmax
- DOUBLE localmin
- DOUBLE localmax
- uint32_t flags
- struct ModbusTransform::@3::@5 range
- union ModbusTransform::@3 u

PointerStack struct Reference

Public Attributes

- Cell *** cells
- int size
- int max
- int top
- int base
- int growrate

PrintContext struct Reference

Public Attributes

- Cell * file
- int32_t english

PT_ChangeRequest struct Reference

Public Attributes

- ThreadMessage tm
- CDataHubPoint * point
- PT_uVALUE value
- PT_TYPE valueType
- int quality

- MSCLOCK timestamp
- struct CTCPCConnection * remoteConnection

PT_stCPOINT struct Reference

Public Attributes

- POINT_COMMON
- void * userdata

PT_uVALUE union Reference

Public Attributes

- ptreal r
- INT64 i
- char * s
- void * v

Scope struct Reference

Public Attributes

- Arg * argrefs
- int maxargs
- int nargs

StackPosition struct Reference

Public Attributes

- int top
- int base

StringStream struct Reference

Public Attributes

- char * buf
- unsigned int end
- unsigned int nextread
- unsigned int nextwrite

- unsigned int allocated
- unsigned int highwrite

Symbol struct Reference

Public Attributes

- Buf name

SymbolMap struct Reference

Public Attributes

- struct Cell * symref

ThreadMessage struct Reference

Public Attributes

- struct ThreadMessage * next
- struct CTCPCConnection * writer
- int id
- int type
- int err
- int flags
- void * payload

TryState struct Reference

Public Attributes

- uint32_t destroying
- uint32_t breaking
- uint32_t returning
- uint32_t throwing
- StackPosition env_stack_pos
- StackPosition function_stack_pos
- StackPosition variable_stack_pos
- StackPosition protect_stack_pos
- Cell * current_env

Type struct Reference

Classes

Public Attributes

- uint32_t flags
- int32_t simple_type
- struct Cell * complex_type
- union Type::@7 t

UT_stCMD struct Reference

Public Attributes

- char * name
- int minargs
- int maxargs
- UT_pfCMD pfunc
- char * description
- int32_t user_i1
- int32_t user_i2
- void * user_v1

ValueStack struct Reference

Public Attributes

- Cell ** cells
- int size
- int max
- int top
- int base
- int growrate

Vector struct Reference

Public Attributes

- int length

- int maxlength
- unsigned short growsize
- unsigned short indexed
- unsigned short isDictionary
- struct Cell ** values
- int * idx

WriteContext struct Reference

Public Attributes

- Cell * file
- Cell ** refarray
- int arrsize
- int arrused
- short curcol
- short width
- short indent
- short pretty
- short english
- short notrace
- short depth
- short maxdepth